# Instructor's Guide and Solutions Manual to accompany

# An Introduction to Formal Languages and Automata

Third Edition

Peter Linz

JONES AND BARTLETT COMPUTER SCIENCE

# Instructor's Guide and Solutions Manual to Accompany

# An Introduction to Formal Languages and Automata

## *Third Edition*

JONES AND BARTLETT PUBLISHERS
*Sudbury, Massachusetts*
BOSTON TORONTO LONDON SINGAPORE

*World Headquarters*
Jones and Bartlett Publishers
40 Tall Pine Drive
Sudbury, MA 01776
978-443-5000
info@jbpub.com
www.jbpub.com

Jones and Bartlett Publishers Canada
2406 Nikanna Road
Mississauga, ON L5C 2W6
CANADA

Jones and Bartlett Publishers International
Barb House, Barb Mews
London W6 7PA
UK

Copyright © 2001 by Jones and Bartlett Publishers, Inc.

ISBN 0-7637-1646-4

All rights reserved. No part of the material protected by this copyright notice may be reproduced or utilized in any form, electronic or mechanical, including photocopying, recording, or any information storage or retrieval system, without written permission from the copyright owner.

Printed in the United States of America
04 03 02 01 00 10 9 8 7 6 5 4 3 2 1

# Preface

My aim in this manual is to provide assistance to the instructor teaching a course based on my book *An Introduction to Formal Languages and Automata*, Third Edition. In offering my advice, I do not intend to suggest what topics to cover, what to emphasize and what to leave out, or what to do on a day-by-day basis. The material in the text is organized in such a way that it leaves little room for significant deviations. A few sections can be omitted, while others can be treated lightly, but on the whole, the material should be covered as it is presented in the book. One may want to devote more time to some topics while spending less on others, but this is a matter of individual taste and should be tailored to the needs and capabilities of the audience. I have little useful advice to contribute to this.

But, as I stated in the preface, the book was organized around one major idea: that this subject matter can be made profitable and enjoyable for the average undergraduate student if mathematical formalism is downplayed and problem-solving is made the focus. This means that students

learn the material and strengthen their mathematical skills primarily by doing problems. Now this may seem a rather obvious idea; all textbooks contain exercises that are routinely assigned to improve and test the students' understanding. But what I have in mind here goes a little deeper. I consider assigned problems not merely a supplement to the lecturers, but a very major part of the course, and believe that to some extent the lectures themselves should be a preparation for the homework. This means that one needs to emphasize those things that will be most helpful in this, with the basic notions in as simple a light as possible, and with good illustrative examples. Lengthy proofs, unnecessary detail, or excessive mathematical rigor have no place in such an approach. Many details and secondary topics are not covered in class, but are relegated to homework exercises. This means that the homework has to be tightly integrated into the lectures and that each problem has some specific pedagogical purpose. Homework assignments need to be composed thoughtfully to achieve a well-understood purpose. This is a difficult task, but in my experience, the success of the course depends critically on how well it is done.

There are several types of exercises, each having a particular purpose and flavor. Some of them are what I characterize as straightforward or drill exercises. By this I mean that any student with a basic understanding should be able to do them. Most of the time they are not very interesting, but they serve a purpose. They test the student's grasp of the material, uncover possible misunderstandings, and give almost everyone the feeling of being able to do something.

A second type of exercise I call "fill-in-the-details." These are usually omitted parts of proofs or examples whose broad outline is sketched in the text. Conceptually these are not difficult since all nonobvious points have already been spelled out. For mathematically well-trained students they are simple. For those not in this category (e.g., many computer science undergraduates), they may be a little more difficult and are likely to be unpopular with them. They are useful primarily in sharpening mathematical reasoning and formalizing skills.

A third type of exercise deals with extension of the material. In the text I cover only what I consider absolutely fundamental and essential, leaving out many interesting ideas and applications. Most of the exercises of this type do not have neat and simple answers (so they will be hard to grade) and are generally open ended. But they are useful because they get the student thinking about the material in a deeper way and possibly reading some more advanced material. This kind of exercise is most appropriate for the brighter student.

Finally, there are the exercises that I call problems. These are a little like puzzles, ranging from the easy to the fairly difficult, but all having the common feature that they take the student at least one step beyond the

obvious. Some of the difficult ones require tricks that may be quite hard to discover. In general, I do not give hints, but you may want to consider doing so on occasion. It is possible for some students, especially the serious ones, to spend many frustrating hours on such problems. The ultimate reward of course can be a great deal of satisfaction, but one has to judge how much this is worth. On the whole, this is the most satisfying kind of exercise for the instructor as well as for the students.

The major new feature of the third edition is the inclusion of a small set of solved exercises. These exercises were selected to help students test their understanding and to provide help in some of the harder problems. Experience has shown that in many difficult cases, students need hints that point them in the right direction or that show how someone else approaches the problem. In such cases, I have selected a representative example from a group, for which the solution will suggest possible attacks on similar problems. I hope that providing my answers will be instructive and help to develop the problem-solving ability of the reader.

I have also made a very rough classification of exercises by starring those that I consider very difficult and double-starring those that are unusually open-ended. But as I find it unproductive to reduce most problems to such a simplistic classification, this is quite restricted. The discussion in the body of this guide is generally much more informative for exercise selection.

# Contents


Chapter 1 . . . . . 1
Chapter 2 . . . . . 5
Chapter 3 . . . . . 13
Chapter 4 . . . . . 19
Chapter 5 . . . . . 26
Chapter 6 . . . . . 30
Chapter 7 . . . . . 34
Chapter 8 . . . . . 40
Chapter 9 . . . . . 44
Chapter 10 . . . . . 46
Chapter 11 . . . . . 51
Chapter 12 . . . . . 54
Chapter 13 . . . . . 58
Chapter 14 . . . . . 62

# Chapter 1

## Section 1.1

The material in this section is assumed as a prerequisite for the course. The exercises are all of the type done in a discrete mathematics course. If students are comfortable with this material, one or two of these can be assigned as refresher or warm-up exercises. If students are struggling with this material, extra time will be needed to remedy the situation. Working out some of these exercises in class and reading the solved exercises should help.

**1 to 12:** These exercises are all fairly simple, involving arguments with sets. Most of them can be solved by direct deduction or simple induction. Exercise 8 establishes a result that is needed later.

**13 to 19:** Material on order of magnitude is included for completeness and because it is needed in the last chapter. If you don't plan to cover complexity, these simple exercises can be skipped.

**20 to 22:** Some routine exercises to remind students of the terminology of graphs.

**23 to 25:** Good exercises in induction. Most students will have seen something close to this in their discrete math course and should know that induction is the way to go.

**26 and 27:** Simple, but good examples of using proof by contradiction.

**28:** (a) and (c) are true and can be proved by contradiction. (b) is false, with the expression in Exercise 27 a counterexample.

**29 and 30:** Classic examples that should be known to most students.

**31:** This is easier than it looks. If $n$ is not a multiple of three, then it must be that either $n = 3m + 1$ or $n = 3m + 2$. In the first case, $n + 2 = 3m + 3$, in the second $n + 4 = 3m + 6$.

## Section 1.2

In this section we introduce the very basic concepts on which the rest of the book is based. One could leave the section out, since everything recurs later. But I think it is important to put this up front, to provide a context for later more specific development. Also, it gives students an immediate introduction to the kinds of problems they will face. There are some reasonably difficult and challenging exercises here.

**1 to 3:** Like Example 1.1, these are all obvious properties of strings and we simply ask to make the obvious rigorous. All can be done with induction and are useful for practicing such arguments in a simple setting. The results are needed again and again, so it is useful to do some of these exercises.

**4:** A simple drill exercise that introduces the idea of parsing (without specifically using the term) and shows that breaking a structure into its constituent parts requires some reasoning.

**5:** Simple, but gets students to think about the meaning of the closure of a language. True for 1.13, false for 1.12.

**6:** An exercise in understanding notation. There are of course no such languages, since $L^*$ and $\overline{L}^*$ both contain $\lambda$.

**7 to 9:** These are not difficult, but require careful reasoning, sometimes involving several steps. The exercises are good tests of the understanding of concatenation, reversal, and star-closure of languages.

**10:** To get the grammars should be easy, but giving convincing arguments may prove to be a little harder. In fact, expect students to ask, "What do you mean by convincing argument?" and you will need to set standards appropriate to your class at this point. It is important that the issue of how much rigor and detail you expect is settled as soon as possible.

**11:** It should not be hard to see that the answer is $\{(ab)^n : n \geq 0\}$.

**12:** Points out that grammar does not have to derive anything, that is, it derives $\varnothing$.

**13:** A mixed bag of exercises. (a), (b), and (c) are easy; so is (d) but it does give trouble to students who don't see that the language is actually $\{a^{m+3}b^m : m \geq 0\}$. Parts (e) to (h) let the students discover how to combine grammars, e.g. if $S_1$ derives $L_1$ and $S_2$ derives $L_2$, then $S \to S_1S_2$ combined with the grammars for $L_1$ and $L_2$ derives $L_1L_2$. This anticipates important results for context-free grammars. Part (i) cannot be done this way, but note that $\overline{L_1 - L_4} = L_1 \cap L_4 = \varnothing$.

**14:** (a) is simple, the others get progressively harder. The answers are not too difficult if students are comfortable working with the mod function. For example, the solution to (c) can be seen if we notice that $|w| \bmod 3 \neq |w| \bmod 2$ means that $|w| \neq 6n$ or $|w| \neq 6n+1$. A grammar then is

$$S \to aaA \,|aaaA|\, aaaaA \,|aaaaaA|$$
$$A \to aaaaaaA|\lambda.$$

**15:** This simple exercise introduces a language that we encounter in many subsequent examples.

**16:** A set of fairly hard problems, which can be solved by the trick of counting described in Example 1.12. (c) is perhaps the most difficult.

(b) $S \to aS \,|S_1S|\, aS_1$.

where $S_1$ derives the language in Example 1.12.

(c) $S \to abSa \,|aaSb|\, bSaa \,|SS|\, \lambda$.

(d) Split into cases $n_a(w) = n_b(w) + 1$ and $n_a(w) = n_b(w) - 1$.

**17:** While conceptually not much more difficult than Exercise 16, it goes a step further as now we need to be able to generate any number of $c$'s anywhere in the string. One way to do this is to introduce a new variable that can generate $c$'s anywhere, say

$$C \to cC|\lambda$$

and then replace every terminal $a$ by $CaC$ in the productions of Exercise 16.

**18:** A fill-in-the-details exercise for those who stress making arguments complete and precise.

**19 and 20:** These examples illustrate the briefly introduced but important idea of the equivalence of grammars. It is important that the students realize early that any given language can have many grammars. The two grammars in Exercise 19 are not equivalent, although some will claim that both generate $\{a^n b^n\}$, forgetting about the empty string. The exercise points out that the empty string is not "nothing." In Exercise 20, note that $S \Rightarrow SSS$ can be replaced by $S \Rightarrow SS \Rightarrow SSS$, so the grammars are equivalent.

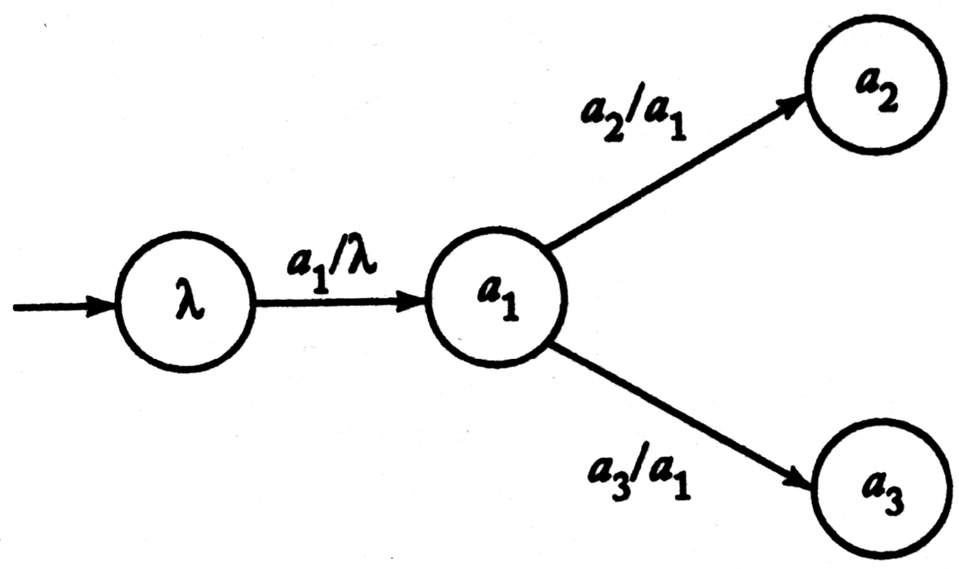
**21:** An exercise to point out that, in general, the left side of a production can be more complicated than a single variable. It will also drive home the point that such more complicated grammars are very difficult to work with. The language derived by the grammar is $\{a^{2n} : n \geq 1\}$, but don't expect many students to see this at such an early stage.

## Section 1.3

Most of the exercises in this set are simple. This section is optional; its purpose is to hint at some applications and perhaps relate to something in the student's previous experience. It also introduces finite automata in an informal way. Exercises 1 to 5 are suitable for those with a background and interest in programming languages. Exercises 9 to 12 deal with some fundamental hardware issues familiar to most computer science students from a course in computer organization. Exercises 7 and 8 may appeal to engineering students with some digital design background. Math majors may not care for any of this, and may be excused from this set. The material plays no significant role in later discussion.

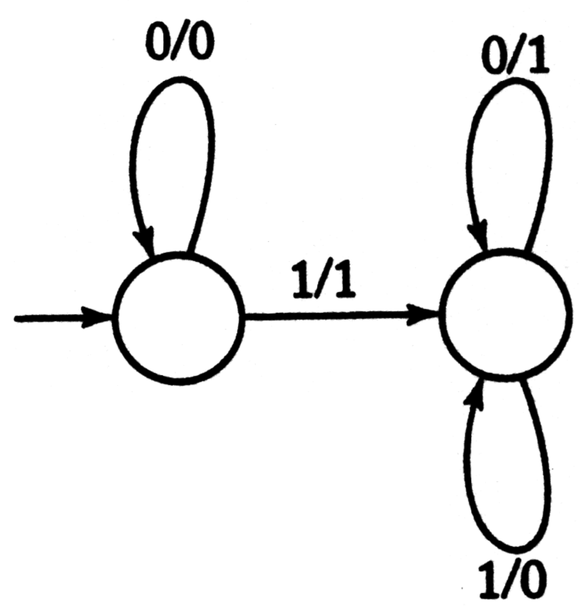
**6:** A more prosaic way of stating the problem is: no $M$ can follow the first $D$, no $D$ can follow the first $C$, etc. The resulting automaton is a little large, but easy in principle.

**7:** This introduces an important idea, namely how an automaton can remember things. For example, if an $a_1$ is read, it will have to be reproduced later, so the automaton has to remember. This can be done by labeling the state with the appropriate information. Part of the automaton will then look like




**8:** A simple extension of the idea in Exercise 7.

**9:** The automaton must complement each bit, add one to the lower order bit, and propagate a carry. Students will probably need to try a few examples by hand before discovering an answer such as the one below.




**10:** A fairly simple solved exercise.

**11 to 13:** These are similar in difficulty to Exercise 10. They all require that the automaton remember some of the previously encountered bits.

**14:** This is a simple exercise as long as the higher order bits are given first. Think about how the problem could be solved if the lower order bits are seen first.

# Chapter 2

## Section 2.1

**1:** A drill exercise to see if students can follow the workings of a dfa.

**2:** Some of these are the same as Exercise 10, Section 1.2, so the student can see the parallels and differences between grammars and automata solutions. Since this is a very fundamental issue, this is a good introductory problem. All the exercises are easy.

**3 and 4:** These two exercises let the student discover closure of regular languages under complementation. This is discussed later in the treatment of closure, so this gives a preview of things to come. It also shows that the dfa for $\overline{L}$ can be constructed by complementing the state set of the dfa for $L$. The only difficulty in the exercises is that they require formal arguments. The arguments are straightforward, so it is a good exercise for practicing mathematical reasoning.

**5:** An easy exercise.

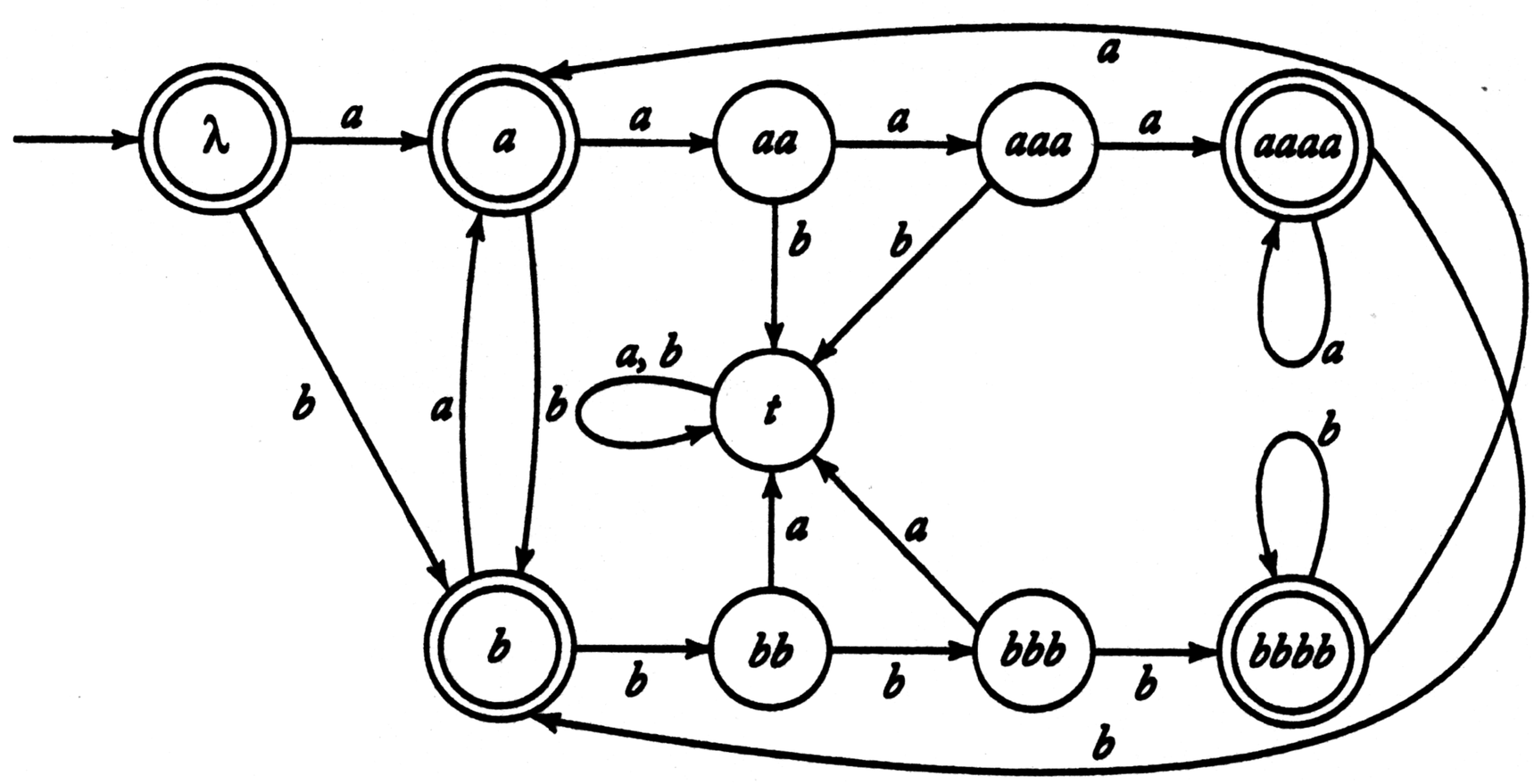
**6:** A little hard to see, but contrasts the precise description of a language via set notation and via a dfa with the ambiguous, but intuitive, verbal ones. The answer in set notation is

$$L = \{b^n a : n \geq 0\} \cup \{b^n awb : n \geq 0, w \in \{a,b\}^*\}$$

which can be stated as: a string must have at least one $a$ and if it has more than one $a$, it must be terminated by a $b$.

**7:** Very similar to Exercise 14 in Section 1.2. The answers all involve simple modular counting by an automaton. Once students grasp this principle, all parts are easy.

**8:** May be quite hard until the student gets into the habit of using mnemonic labels. If each state is labeled with the appropriate number of $a$'s, the solution for part (a) below follows directly. Solutions to the other two parts are comparable in difficulty.

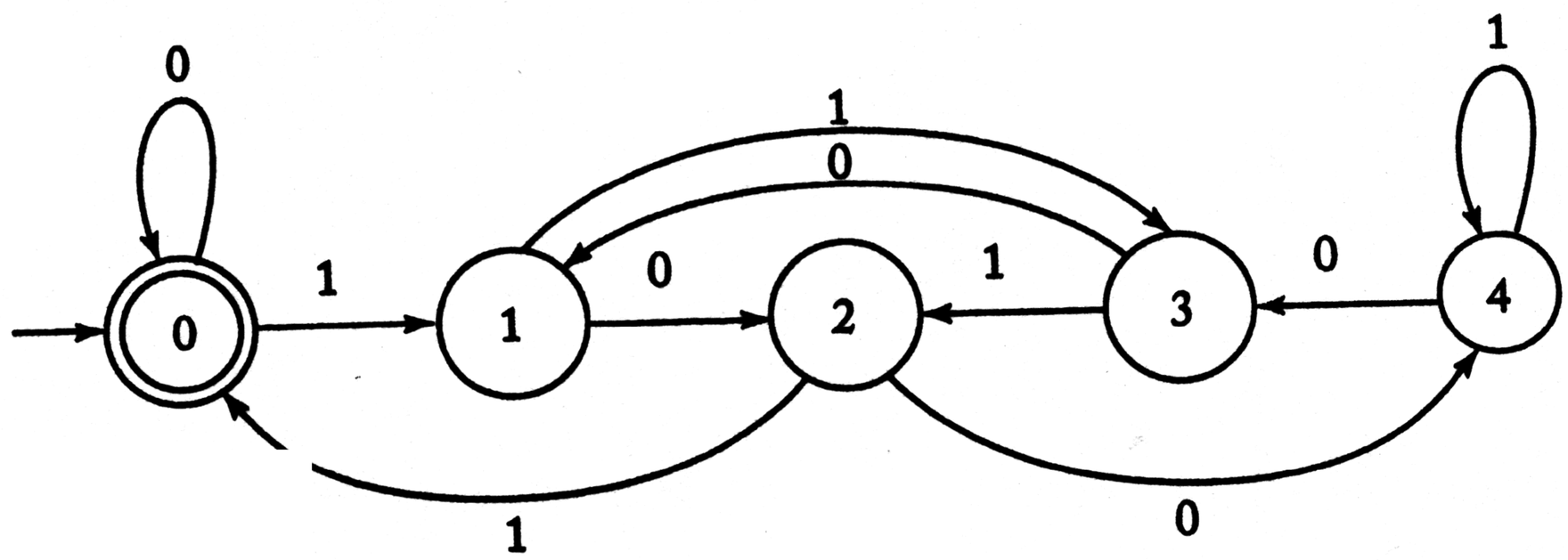



**9:** Continues the theme of Exercise 8, but is on the whole a little easier. After this exercise, the students should be convinced of the usefulness of mnemonic labeling. Note that (a) and (b) are not the same problem.

**10:** This is a difficult problem for many students. They will try to find some kind of repeated pattern. The trick is to label the states with the value (mod 5) of the partial bit string and find the rule for taking care of the next bit by

$$(2n+1) \bmod 5 = (2n \bmod 5 + 1) \bmod 5$$

leading to the solution shown below.




Don't expect everyone to discover this, so a hint may be appropriate (or let them try a bit first before giving directions).

**11 to 14:** All fairly easy exercises, reinforcing the idea that a language is regular if we can find a dfa for it.

**15:** A simple problem, pointing out an application in programming languages.

**16:** This exercise introduces the important technique of a general construction. Given any dfa for $L$, how do we construct from it a dfa for $L - \{\lambda\}$? The temptation is to say that if $\lambda \in L$, then $q_0$ must be in the final state set $F$, so just construct a new automaton with final state set $F - \{q_0\}$. But this is not correct, since there may be a nonempty string $w \in L$, such that $\delta^*(q_0, w) = q_0$. To get around this difficulty, we create a new initial state $p_0$ and new transitions

$$\delta(p_0, a) = q_j$$

for all original

$$\delta(q_0, a) = q_j.$$

This is intuitively reasonable, but it has to be spelled out in detail, so a formal argument will still be hard. However, as it is one of the simplest cases of justifying a construction, asking for a proof that the construction works is a good introduction to this sort of thing.

**17:** A good exercise in inductive reasoning as well as in handling concise, but not very transparent, mathematical notation.

**18 and 19:** These involve generalization of the idea introduced in Example 2.6 and should not prove too difficult.

**20:** Generalizes the above idea in Exercises 17–19 a little more and points to closure properties to come. Gets the student to think ahead about issues that will be treated in more detail later. Not too hard for students with the ability to generalize, although the formal proof may be challenging.

**21:** An exercise for reasoning with transition graphs. The answer is intuitively easy to see, but may be troublesome if you are asking for a formal proof.

**22:** Constructions of this type are very fundamental in subsequent discussions, but at this point the student has no experience with them, so this will be hard. But if the student can discover the idea behind the solution, later material will make much more sense. Perhaps a detailed hint such as "if $va \in L$, then $\delta^*(va) \in F$. But $v \in truncate(L)$, so that $\delta^*(v)$ must be a final state for the new automaton" is worthwhile. This will probably give the construction away, but any kind of formal proof will still be hard for most.

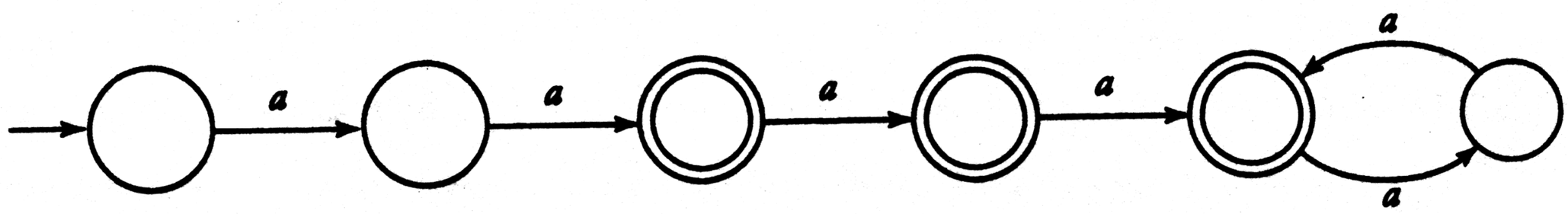
**23:** Not hard, but introduces an alphabet whose elements are triples.

**24:** A simple exercise that has many different solutions.

## Section 2.2

**1:** A "fill-in-the-details" exercise, of which there are a number throughout the text. Such exercises tend to be unexciting to many students and you may not want to assign a lot of them. An occasional one, though, is appropriate. Having to reason about fine points gives the student a better understanding of the result. It will also test the students' ability to go from an intuitive understanding of a proof to a precise sequence of logical steps.

**2:** An exercise foreshadowing the dfa/nfa equivalence. An answer such as




is not entirely trivial for students at this point.

**3 to 5:** Routine drill exercises in understanding and following transition graphs. Also reinforces the point that $\delta^*$ is a set.

**6 and 7:** These require solutions with a bound on the number of states. Without such bounds the exercises would be trivial, but even with them they are not too hard. The main virtue of this set is that it gets the student to play around with various options. A solution to 6 is easy. Exercise 7 is solved.

**8:** The answer is pretty obvious, but how can one defend such a conclusion reliably? A question without a very tidy answer, but it gets the student to think.

**9:** This should be quite manageable for students who have studied the solved Exercises 7 and 8.

**10:** An easy exercise.

**11:** A routine, but worthwhile exercise, since some students get confused tracing through an nfa.

**12:** Clearly, the language accepted is $\{a^n, n \geq 1\}$. Assuming $\Sigma = \{a\}$, the complement is $\{\lambda\}$.

**13:** A very easy exercise when nondeterminism is understood.

**14:** A little hard to see, but guaranteed to make students think about nondeterminism. A simple answer: 0, 1, and all strings of length two or greater, terminated by 1 or 00.

**15:** Can be a little hard, mainly because of the unusual nature of the question. It is easy to come up with an incorrect result. For the solution, see the solved exercises.

**16:** Again the obvious answer is no, but this is not so easy to defend. One way to argue is that for a dfa to accept $\{a\}^*$, its initial state must be a final state. Removing any edge will not change this, so the resulting automaton still accepts $\lambda$.

**17:** A worthwhile exercise about a generalization of an nfa, which is really nothing new. Students sometimes ask why in the definition of a finite automaton we only have one initial state, but may have a number of final states. This kind of exercise shows the somewhat arbitrary nature of the definition and points out that this arbitrariness is inconsequential. To solve the exercise, introduce a new initial state $p_0$ and $\lambda$-transition $\delta(p_0, \lambda) = \{q_0\}$, for all $q_0 \in Q_0$.

**18:** No, in $q_0 \in F$, introduce $p_0$ as in the exercise above.

**19:** Exercise in reasoning with transition graphs. Makes sense intuitively, but don't expect everyone to produce an airtight argument.

**20:** This introduces a useful concept, an incomplete dfa (which some authors use as the actual definition of a dfa). Using incomplete dfa's can simplify many problems, so the exercise is quite worthwhile.

## Section 2.3

**1:** Straightforward, drill exercise. For a simple answer, note that the accepted language is $\{a^n : n \geq 1\}$.

**2 and 3:** These are easy drill exercises.

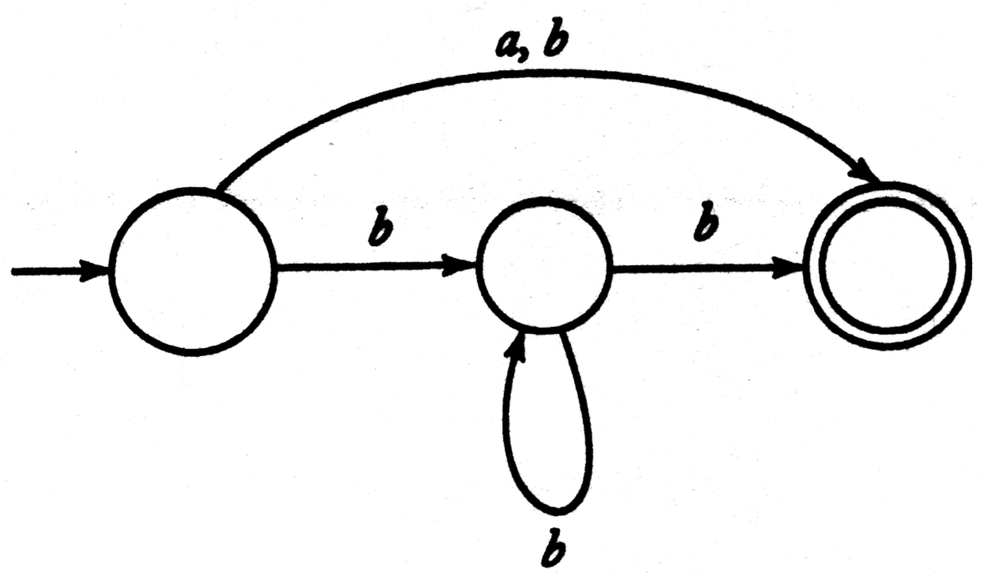
**4:** A "fill-in-the-details" exercise to supply some material omitted in the text.

**5:** Yes, it is true. A formal argument is not hard. By definition, $w \in L$ if and only if $\delta^*(q_0, w) \cap F \neq \varnothing$. Consequently, if $\delta^*(q_0, w) \cap F = \varnothing$, then $w \in \overline{L}$.

**6:** Not true, although some students will find this counterintuitive. This exercise makes a good contrast with Exercise 5 above and Exercise 4, Section 2.1. If the students understand this, they are on their way to understanding the difficult idea of nondeterminism.

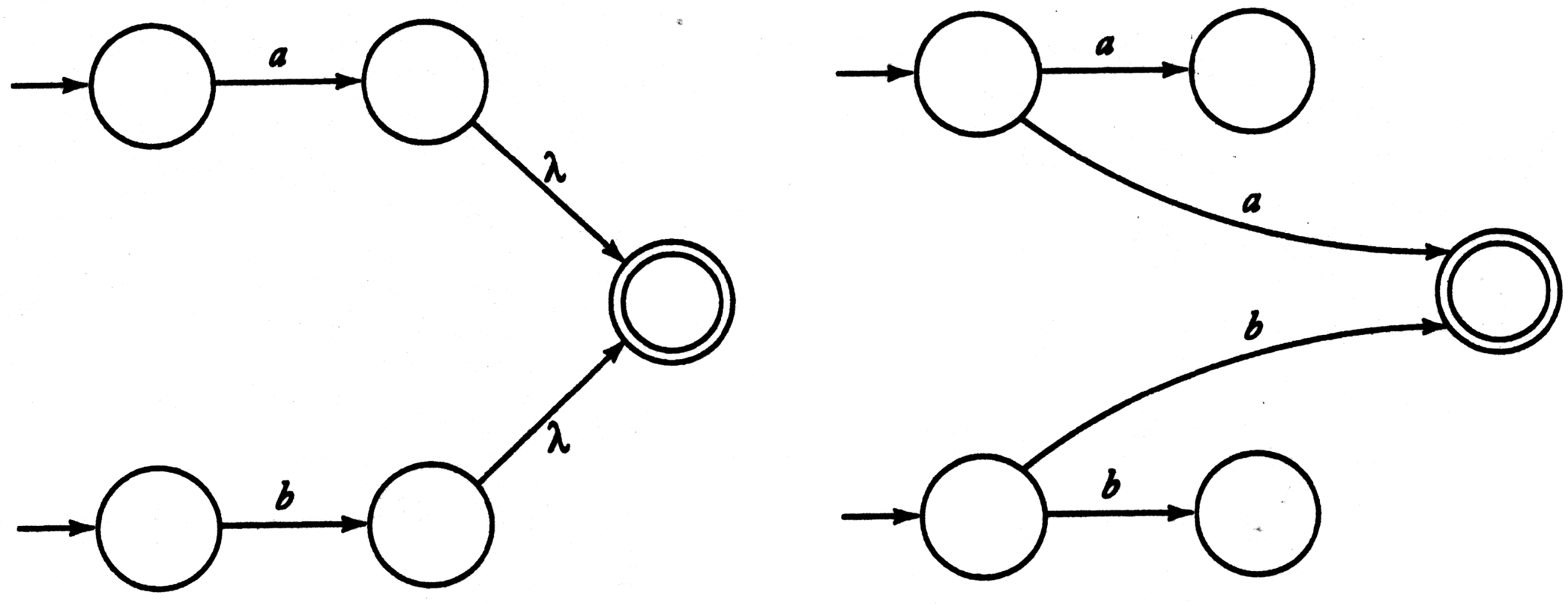
**7:** Create a new final state and connect it to the old ones by $\lambda$-transitions. This does not work with dfa's; to convince yourself of this, try to get a dfa with one final state that accepts $\{a, bb\}$.

**8:** Does not follow from Exercise 7 since $\lambda$-transitions are forbidden. The answer requires some thinking. One answer is shown below.




This is a specific case of the general construction in the next exercise.

**9:** This is a troublesome construction. Start with dfa, add a single final state with $\lambda$-transitions, then remove the $\lambda$-transitions as sketched below.




**10:** Another hard one. Introduce a single initial state, and connect it to previous ones via $\lambda$-transitions. Then convert back to a dfa and note that the construction of Theorem 2.2 retains the single initial state.

**11:** An instructive and easy exercise, establishing a result needed on occasion. Without this exercise some students may not realize that all finite languages are regular.

**12:** Another exercise foreshadowing closure results. The construction is easy: reverse final and initial states, then use the conclusion of Exercise 13, Section 2.2.

**13:** Once you see that the language is $\{0^n : n \geq 1\} \cup \{0^n 1 : n \geq 0\}$, the problem is trivial.

**14:** A difficult problem with a solution in the solved exercises.

**15:** Not obvious, but should be manageable after one or two similar problems. Find the set of states $Q_2$ for which there is a path of length two from the initial state. Introduce a new initial state $\widehat{q_0}$ and $\lambda$-transitions from $\widehat{q_0}$ to the states in $Q_2$.

## Section 2.4

**1:** This is an easy drill exercise.

**2:** Easy to construct dfa's; the rest is routine.

**3:** This is an important point that may have escaped notice: in minimization, the original and final automata must be dfa's. So things will work as stated only if *reduce* produces a dfa. This is the case because each $i$ can occur in only one label of $\widehat{M}$.

**4:** Another easy drill.

**5:** Every walk from an initial to a final state must have length at least $n$. Thus all simple paths have length at least $n$, and consequently there must be at least $n+1$ vertices.

**6:** Not obvious, but the solution is given.

**7:** A test of students' ability to work with equivalence relations.

**8:** This exercise asks the student to fill in some of the details that were omitted in the text.

**9:** Personally, I am not in favor of programming exercises for this course. Computer science students latch on to programming as a kind of security blanket, and if you assign too much of it, they may get the wrong idea of what the course is about. But for those who feel differently, I include an occasional exercise of this kind. They can be abbreviated a bit by asking for a pseudo-code solution instead of a full-blown program. In any case, it is useful for the student to realize that many of the constructions in the text make useful algorithms.

Although a little lengthy, this is a good exercise; it will certainly clear up any misunderstanding of the algorithms in this section. If you ask for a complete implementation, students will have to think about how to represent a dfa in a computer program.

**10:** Again, a test of understanding equivalences involving a short proof by contradiction.

**11:** "Fill-in-the-details" and an exercise in constructing partitions.

# Chapter 3

## Section 3.1

The material in this section lends itself very well for problem-solving exercises. The ones given here range from simple to moderately difficult. It is easy to make up similar problems, but one needs to be careful. Some very innocent looking questions can be quite difficult. It appears to be harder for most students to work with regular expressions than with finite automata.

**1:** A routine drill exercise.

**2:** Yes.

**3:** Since $1^*$ includes $\lambda$, this is also true.

**4:** Easy, once you split this up into ($n$ and $m$ both even) or ($n$ and $m$ both odd). An answer is

$$(aa)^* (bb)^* + a (aa)^* b (bb)^* .$$

**5:** (a) and (b) are easy, since the solution can be written down immediately, e.g.

$$r = aaaaa^* (\lambda + b + bb + bbb) .$$

To solve (d), it is best to split it into several cases, such as $n < 4$, $m \leq 3$, $n \geq 4$, $m > 3$, etc. We must also include strings in which an $a$ can follow a $b$.

**6:** Uncovers some notational subtleties that may have escaped attention. Particularly, the concatenation of a language with the empty set is often misunderstood (of course, it is not really a very important point). Answers: $(\varnothing^*)^* = \{\lambda\}$, $a\varnothing = \varnothing$.

**7:** All strings of the form $w_1 b w_2$, where $w_1$ and $w_2$ are composed of an even number of $a$'s, or $w_1 b a w_2$, where $w_1$ and $w_2$ consist of an odd number of $a$'s.

**8:** Good exercise to see how to get $L^R$ by reversing the regular expression. Leads to the more general question in Exercise 19.

**9:** Split into three cases: $n \geq 1$, $m \geq 3$; $n \geq 2$, $m \geq 2$; and $n \geq 3$, $m \geq 1$.

**10:** Easy problem, with answer $r = abbbb^* (a + b)^*$.

**11:** A little hard to see. The part of getting an odd number of $a$'s or an even number of $b$'s is easy. But we also must include strings where an $a$ can follow a $b$. An answer is $(a+b)^* ba (a+b)^* + a(aa)^* b^* + a^* (bb)^*$, but many quite different expressions are possible.

**12:** Enumerate all strings of length two to get something like $aa(a+b)^* aa + ab(a+b)^* ab + \cdots$.

**13:** Examples 3.5 and 3.6 give a hint. A short answer is $(1+01)^* 00 (1+10)^*$, but students are likely to come up with many different solutions.

**14:** A sequence of simple to harder problems. (a) is easy, (b) a little harder since it should be broken into parts–no $a$'s, exactly one $a$, etc. (e) is quite hard since it involves putting several ideas together, for example runs of $a$'s of length $3n$, separated by substrings with no runs of $a$'s. To get started, look at

$$r = r_1 (baaabr_1baaab)^* r_1,$$

where $r_1$ generates all strings that have no runs of $a$'s. This gives a subset of what we want, but several other parts have to be considered.

**15:** These again increase in difficulty, but on the whole a little harder than Exercise 14. (f) can be rather tedious and frustrating. First look at

$$r = (r_1 100 r_1)^*,$$

where $r_1$ is a regular expression for strings not containing 10. Unfortunately, this does not cover the many special cases, such as $1^*$, and it takes some effort to sort this all out.

**16:** (a) is not hard, (b) perhaps a little harder with answer $b^* + (b^* ab^* ab^* ab^*)^*$. (c) can be solved along these lines, but is longer since it must be split into separate cases $n_a(w) \bmod 5 = 1, 2, 3, 4$.

**17:** Once 16 is solved, this involves putting $c$'s into arbitrary positions.

**18:** Not too important, since these identities will not be used much. They are easy to see but quite hard to justify precisely, so if you assign any, say what level of detail you expect. I accept some arguments like

$$\begin{aligned}(r_1^*)^* &= (\lambda + r_1 + r_1 r_1 + \cdots)^* \\ &= \text{any number of } r_1 \text{ concatenated} = r_1^*.\end{aligned}$$

Not very precise, but good enough to see that the student understands the principle.

**19:** A general question on how to get the reverse of a language through the reverse of its regular expression. It is not hard to see that the only thing we need is to replace $ab \cdots cd$ by $dc \cdots ba$ recursively, but the justification requires a somewhat lengthy argument.

**20:** Formal arguments to provide some omitted details.

**21:** Gets the student to think about the meaning of closure. For a more challenging exercise, omit the disclaimers about $\lambda$ and $\varnothing$.

**22 and 23:** These are simple exercises, but interesting as they point to some advanced applications in pattern recognition. Students may find it interesting to realize that formal languages can be used for something beside describing strings. There is lots in the literature on chain-codes.

**24:** Easy, but introduces the concepts of the next section.

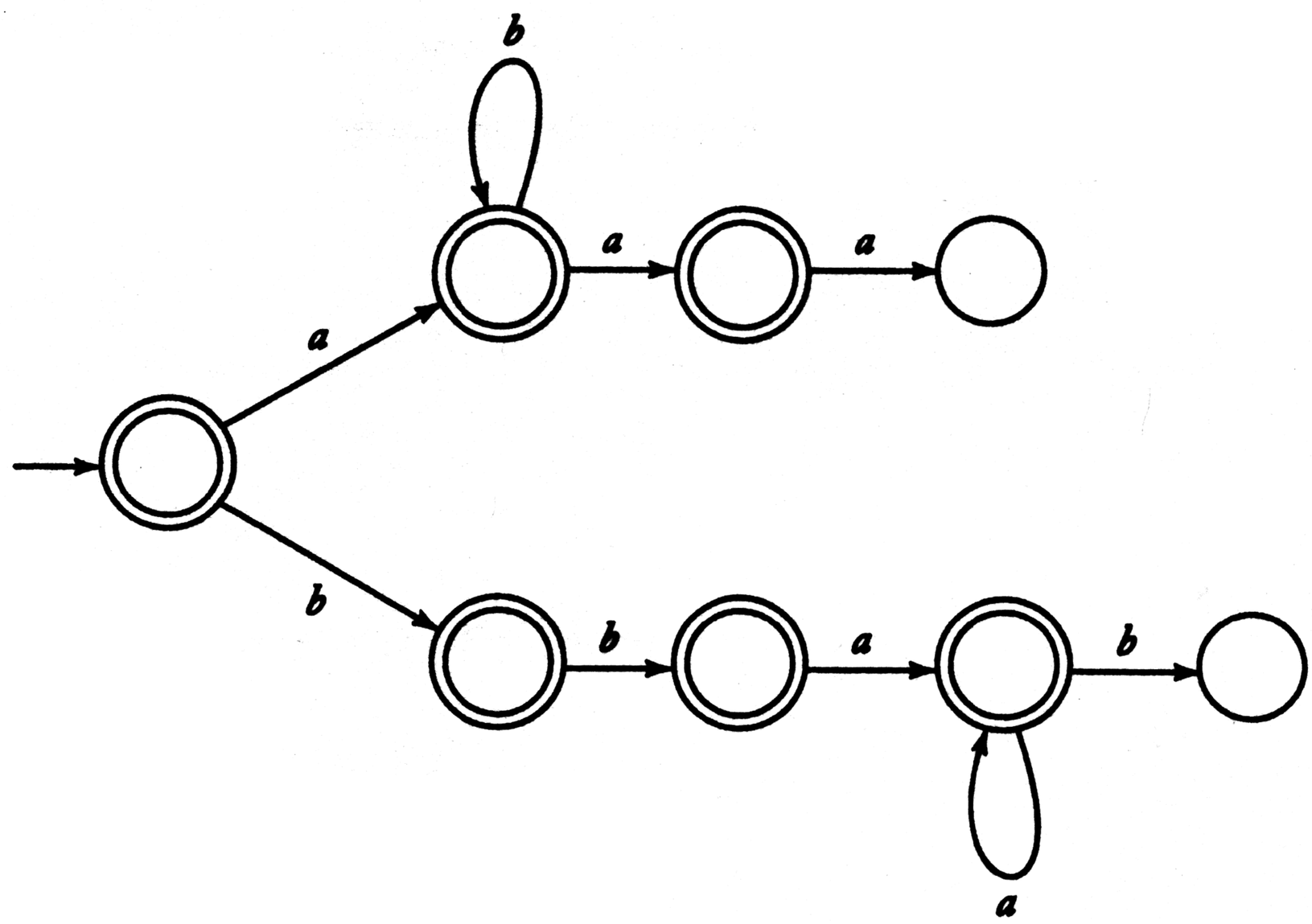
**25:** An exercise with a somewhat different flavor, with a solution provided.

**26:** $r = 1 + 10 + 11 + 100 + 101 + 110 + 111 + 1000 + 1001 + 11111\,(0+1)^*$

## Section 3.2

**1:** Routine application of a given construction.

**2:** Hard if not approached correctly (what is the complement of $L$?), but easy if you look at it the right way. Here the answer can be found by complementing an appropriate dfa. This works, since the most natural construction is a dfa (or at least an incomplete one). A partial answer we get is




with undefined transitions to an accepting trap state.

**3:** Routine drill exercise.

**4:** Here the student can take the tedious route of regular expression to nfa to dfa, but of course they can be done from first principles more easily. Traps those who blindly follow the algorithmic constructions.

**5:** The nfa's are trivial and the nfa-to-dfa constructions routine. Solutions from first principles may be a little cleaner but require more thought.

**6:** Good contrast to Exercise 15(f), Section 3.1. Most students will find it easier to get the finite automaton first, then the regular expression from it.

**7:** Addresses some omitted detail. Straightforward, but a little long.

**8:** Routine application of the construction in Theorem 3.2.

**9:** The answer is easy to see if you note that there is no walk from one of the states to the final state, so that it can be removed.

**10:** Part (a) can be done by inspection. Parts (b) and (c) can be done by the given construction, but we must first create an nfa with a single final state, distinct from the initial state. This is a reminder that the construction in Theorem 3.2 requires an nfa of a special form.

**11:** Just drill.

**12:** Easy if an nfa is constructed first, very hard without this step. This exercise will give trouble to those who try to find regular expressions directly. It also demonstrates that finite automata are easier to work with than regular expressions.

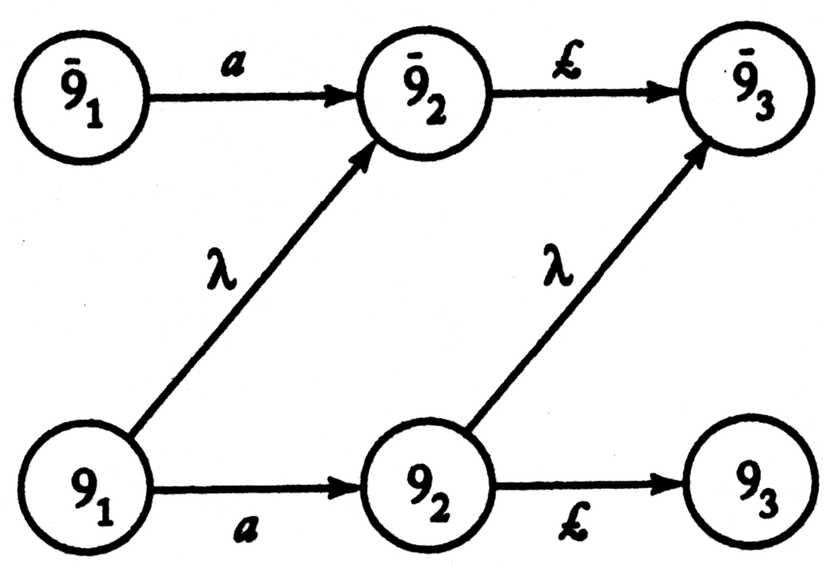
**13:** A somewhat lengthy problem. An nfa is readily constructed as in Exercise 23, Section 2.1. But the most obvious nfa does not satisfy the conditions of Theorem 3.2. After it is put into the right form, we can apply the construction, but it is a little tedious.

**14:** A "fill-in-the-details" exercise which is good practice for proof by induction.

**15 and 16:** Easy, given mainly to keep connection with applications in mind.

**17:** Quite hard, but the given solution should help students in the next exercise.

**18:** The following pattern can be used as the basis for the construction.




**19:** This shows why the simple automata in Figure 3.1 were chosen in their particular way. The construction works even for the unusual situations in this exercise.

## Section 3.3

The exercises in this section are mostly routine, since the constructions relating finite automata to regular grammars are quite automatic.

**1 and 2:** Routine drill exercises.

**3:** The language is $L\left(abba\,(aba)^*\,bb\right)$. From this the left-linear grammar can be found without much trouble.

**4:** Straightforward, with a typical answer $S \to aaA, A \to aA\,|B$, $B \to bbbC, C \to bC|\,\lambda$.

**5:** It is trivial to get a grammar for $L\,(aab^*ab)$. Then we need the closure of this language, which requires only a minor modification. An answer is $S \to aaA, A \to bA\,|C, C \to ab|\,abS$.

**6:** Split into parts (no $a$'s, one $a$, etc.), then get a grammar for each. This is an exercise in combining grammars to get the union of languages.

**7:** Worthwhile exercise of a more formal nature. Proving this should be within the capability of most students. We need to show that whatever string can be derived with $\widehat{G}$, its reverse can be derived with $G$, and vice versa. Typical for this and similar cases, one uses an induction on the length of the string.

**8:** Exchange initial and final states of nfa. This gives an nfa for $L^R$. Get a right-linear grammar from this, then reverse strings in the productions.

**9:** Easy to do from first principles, using the same kind of argument as in Exercise 5.

**11:** This looks worse than it is. Split into two parts: $n_a\,(a)$ and $n_b\,(a)$ are both even, and $n_a\,(a)$ and $n_b\,(a)$ are both odd.

**12:** Constructing a dfa for these languages should by now be easy for most students. After this apply Theorem 3.4.

**13:** Introduces the important idea of a normal form, about which we will say more shortly. The technique here is also important later; we introduce new variables, for example

$$A \to a_1 a_2 B$$

becomes

$$A \to a_1 C$$
$$C \to a_2 B$$

and so on. It is satisfying for the student to discover this!

**14:** If there is no such rule, no sentence can be derived.

**15:** A straightforward, applications-oriented exercise.

# Chapter 4

## Section 4.1

Most of the closure results come either from some construction (such as the one in Theorem 4.1) or from some set identity (as in Example 4.1). The exercises in this section expand on this observations. Keep in mind that some of the constructions can be quite difficult for the students.

**1:** Prove the result by induction on the number of moves of the nfa.

**2:** This is a routine exercise that tests the understanding of the algorithm involved in a constructive proof. It is worthwhile as it gets students used to constructions involving Cartesian products of states.

**3:** Follow the construction for the intersection. The final states of the new automaton are $(q_i, q_j)$, with $q_i \in F$ and $q_j \notin F$.

**4:** Straightforward, fill-in-the-details.

**5:** Good example of simple induction on the number of languages.

**6:** We use

$$S_1 \ominus S_2 = (S_1 \cup S_2) - (S_1 \cap S_2)$$

and known closure properties. A constructive proof (along the lines of intersection in Theorem 4.1) can be made and it may be worthwhile to extend the exercise by asking for such a proof.

**7:** Follows from

$$\text{nor}\,(L_1, L_2) = \overline{L_1 \cup L_2}.$$

**8:** Follows simply from previous closure results.

**9:** Gets the student thinking about homomorphism. (a) and (c) are true, but (b) is false, as shown by the example $L_1 = L\,(a^*)\,, L_2 = L\,(b^*)\,, h\,(a) = a, h\,(b) = a$.

**10:** Easy problem, with answer $L_1/L_2 = L\,(a^*)$.

**11:** A counterexample may not be obvious. Here is one: $L_1 = \{a^n b^n : n \geq 0\}\,, L_2 = \{b^m : m \geq 0\}$. Then $L_1 L_2 / L_2 = \{a^n b^m : n \geq 0, m \geq 0\}$.

**12:** A somewhat unusual and challenging problem. The answer is given.

**13:** The construction can be done either via nfa's or regular expressions. Neither approach should be hard to discover. For example, if $r$ is a regular expression for $L$ and $s$ is a regular expression for $\Sigma$, then $rss$ is a regular expression for $L_1$.

**14:** An easy problem. Since $L$ is regular, so is $L^R$. Then by closure under concatenation, so is the language under consideration.

**15:** While for someone who thoroughly understands the construction of Theorem 4.4 this problem is in principle not too hard, don't assume that this will be easy for your students. The reasoning requires some ability to adapt a difficult argument for a new purpose. We can change the right-quotient argument as follows. We first find all $q_j$ that can be reached from $q_0$ with some string from $L_2$ (e.g., by making each $q_i$ a final state and intersecting the $L_2$). All such reachable states are then made the initial states of an nfa for $L_1 L_2$. This gives an nfa with multiple initial states, but according to Exercise 13 in Section 2.2 this is ok.

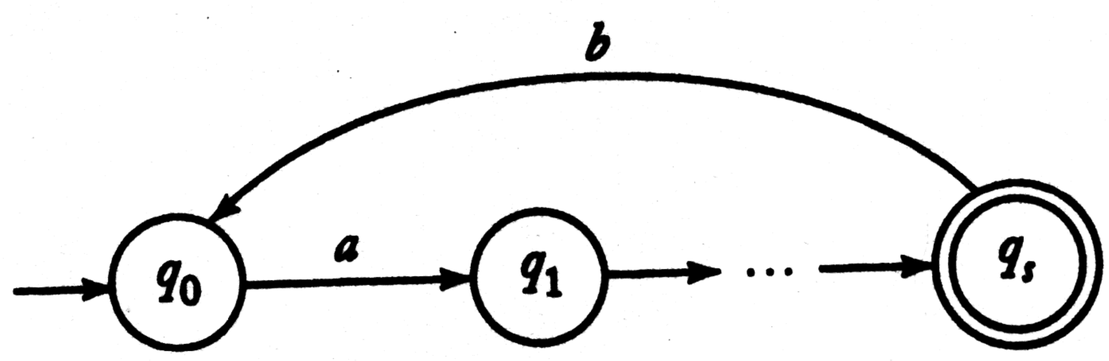
Exercises 17 to 25 all involve constructions of some difficulty. Several of these have been encountered peripherally in previous sections, but if this is the first time students see this type of problem, expect some confusion. However, once a student has mastered one or two, the rest are not so bad. In addition to seeing the construction, there is also the difficulty of proving that the construction works as claimed. This is mostly a matter of using induction on the length of the strings involved, but there are lots of details that will give many students trouble.

Although these are not easy problems, I suggest you assign at least one or two of them. Examining the given solution of 18 should help.

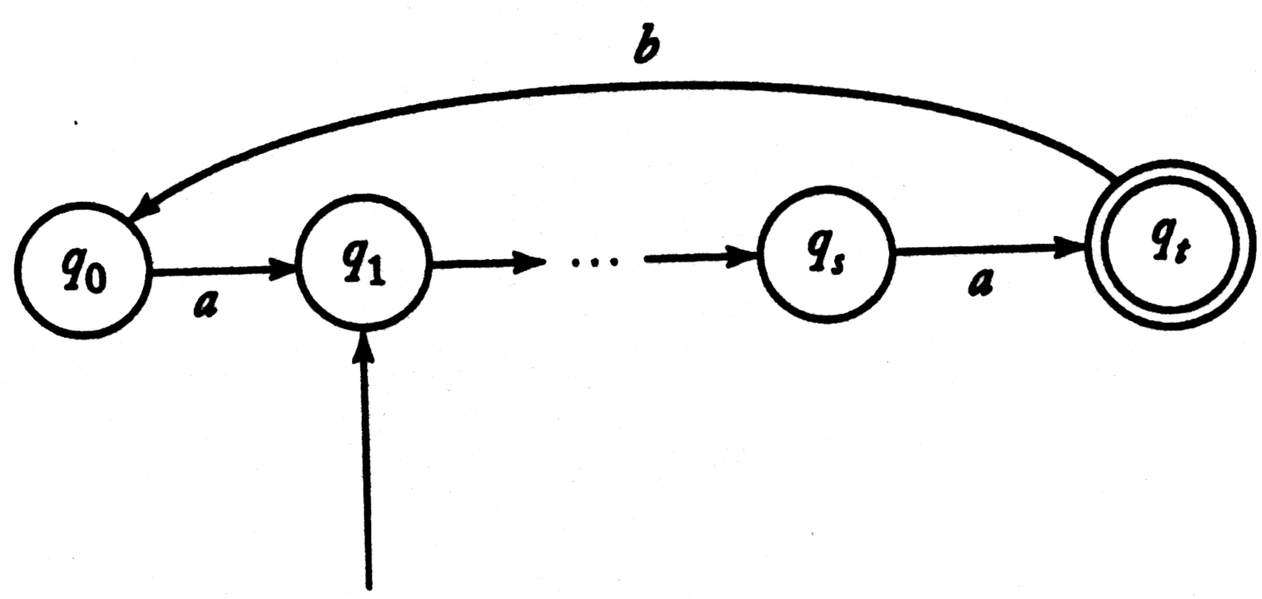
**17:** Find the set of all states $\delta^*(q, w) \in F$ for some $w$. Create a new initial state and add a $\lambda$-transition from it to all elements of $Q$.

**19:** Extend the construction in Exercise 14, Section 2.3. The idea is the same here, but there are a few more details.

**20:** Suppose the graph for $L$ looks like

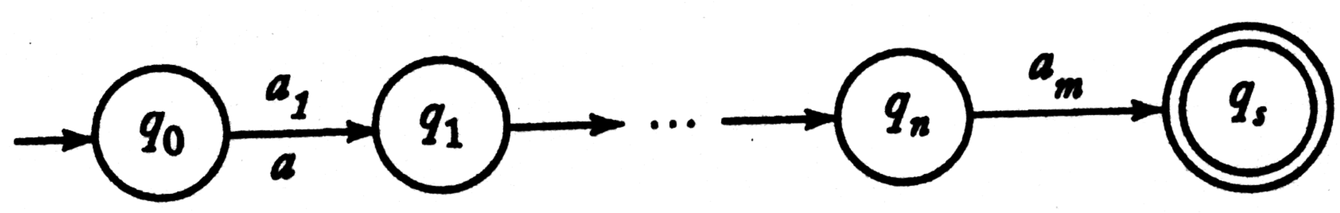



We replace this with

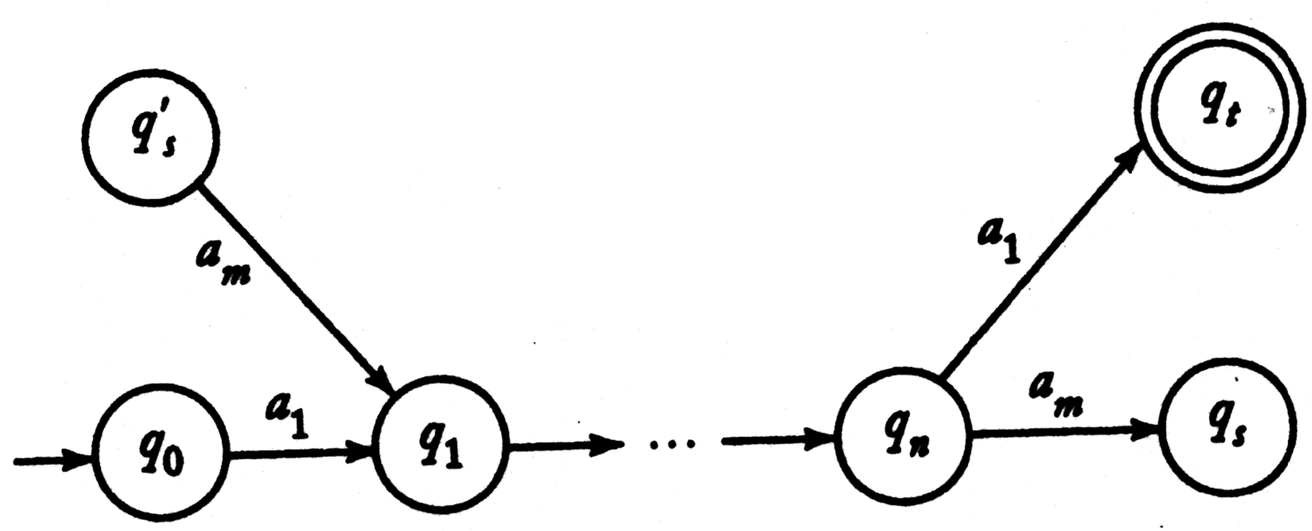



If $q_0$ has several outgoing edges, we create a sub-automaton for each, giving us an nfa with multiple initial states.

**21:** The idea here is similar to Exercise 20. We replace each part of the graph of the form

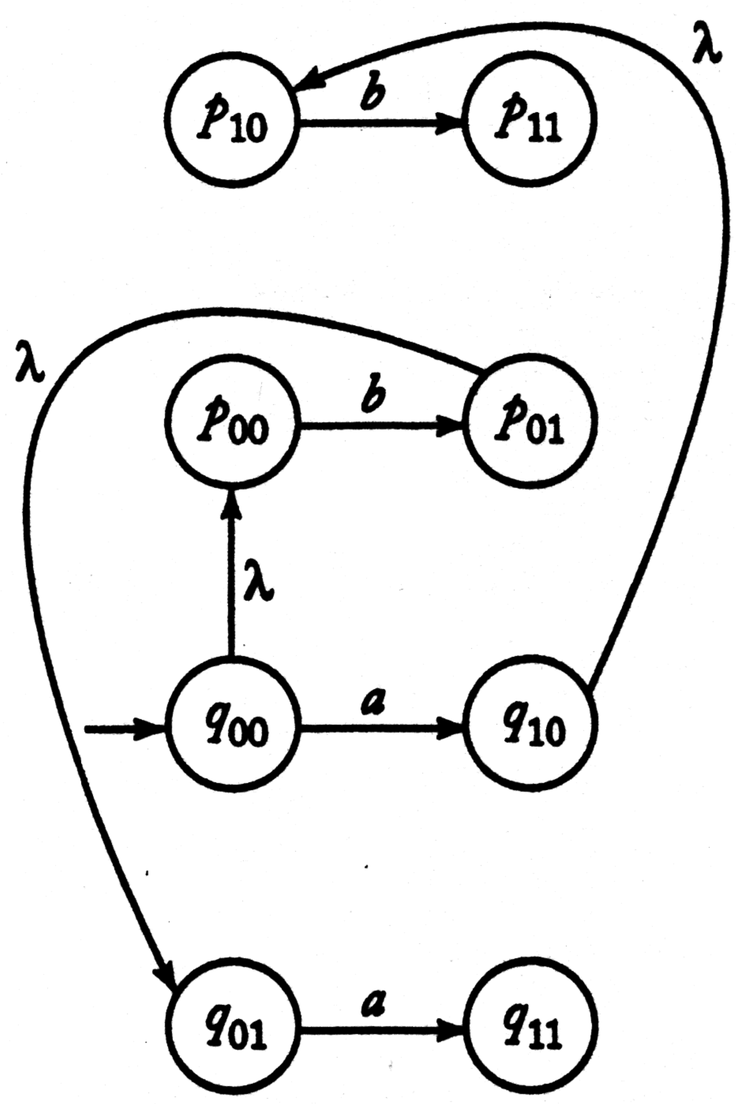



by



but for this to work, the final states of the original automaton cannot have any outgoing edges. This requirement can also be met by introducing new final states and suitable $\lambda$-transitions.

**22:** A difficult construction. We can transfer from the automaton for $L_1$ to that of $L_2$ any time by the addition of suitable $\lambda$-transitions, but the resulting automaton has to remember where to return to. This can be done by creating a sub-automaton of one for each state of the other, specifically, if the automaton for $L_1$ has state set $Q$ and that of $L_2$ has state set $P$, the automaton for $shuffle\,(L_1, L_2)$ will have a state set of size $2\,|Q|\,|P|$ arranged somewhat as in the picture below.




Subscript $i, j$ means that current state is $q_i, p_j$.

**23:** Find all states that can be reached from the initial state in five steps. Then define a new initial state and add $\lambda$-transitions to them.

**24:** For each $q_i \in Q$ of the automaton for $L$, construct two nfa's. The first one has $q_i$ as a final state, the second one has $q_j$ as its initial state. If the languages represented by the first automaton and the reverse of the language corresponding to the second have any common element, then $q_i$ becomes a final state for the automaton for $leftside\,(L)$.

**25:** Take the transition graph of a dfa for $L$ and delete all edges going out of any final vertex. Note that this works only if we start with a dfa!

**26:** Simple view of closure via grammars. The ideas are intuitively easy to see, but the arguments should be done carefully.

## Section 4.2

Most of the exercises in this section can be done constructively by explicitly exhibiting an automaton answering the given questions. But as demonstrated with several examples in previous sections, set operations together with known algorithms are often a quicker way to the solution. If the students realize this, they will find the exercises much easier.

**1:** Simple since $L_1 - L_2$ is regular and we have a membership algorithm for regular languages.

**2:** If $L_1 \subseteq L_2$, then $L_1 \cup L_2 = L_2$. Since $L_1 \cup L_2$ is regular, and we have an algorithm for set equality, we also have an algorithm for set inclusion.

**3:** Construct a dfa. Then $\lambda \in L$ if and only if $q_0 \in F$.

**4:** Since $L_1/L_2$ is regular and we have an algorithm for set equality, we have an algorithm for this problem.

**5:** Use the dfa for $L$, then interchange initial and final states to get an nfa $\widehat{M}$ for $L^R$. Then check if $L(M) = L\left(\widehat{M}\right)$.

**6:** Similar to Exercise 5, except that we need to check if $L(M) \cap L\left(\widehat{M}\right) = \varnothing$.

**7:** Construct the regular language $L_1 L_2$ (e.g. by concatenating their regular expressions), then use the equality algorithm for regular languages.

**8:** Very simple, since $L^*$ is regular.

**9:** This is a little harder than some of the above exercises. Take a dfa for $L$. Then for each pair $(q_i, q_j)$ such that $\delta^*(q_0, u) = q_i$ and $\delta^*(q_j, v) = q_f$, construct an automaton $M_{ij}$ with $q_i$ as initial and $q_j$ as final state. Then determine if $\widehat{w} \in L(M_{ij})$.

**10:** By Exercise 22, Section 4.1, there exists a construction that gives $M_s$ such that $L(M_s) = \mathit{shuffle}(L, L)$. Using the algorithm for equality of regular languages, we can then determine if $L(M_s) = L$.

**11:** Similar to Exercise 10. Construct $M_t$ such that $\mathit{tail}(L) = L(M_t)$.

**12:** A good exercise that involves a simple, but not obvious trick. If there are no even length strings, then

$$L\left((aa + ab + ba + bb)^*\right) \cap L = \varnothing.$$

Some students will come to grief trying to argue from transition graphs.

**13:** Similar to Exercise 12, but now we want to check if $L\left((aa+ab+ba+bb)^*\right)\cap L$ is infinite.

**14:** This is a very simple problem since we have an algorithm to test equality of two regular languages and $\Sigma^*$ is regular.

## Section 4.3

The correct application of the pumping lemma is very difficult for many students. In spite of repeated admonitions, there are always some who will base their arguments on a specific value of $m$ or a decomposition of their own choosing. I don't know how to overcome this except to give lots of exercises.

**1 and 2:** The proofs of these are nearly identical to the proof of Theorem 4.8; we just have to look at the states traversed during the reading of any part of the string. These are very useful exercises, as they make the student think about what is involved in the proof of the pumping lemma. Also, the results are helpful, for example in Exercise 14 later.

**3:** A simple problem: pick $\{a^n b^n\}$ as example. Also, here $L = L^*$.

**4:** This set is generally easy, and involves little more than a routine application of the pumping lemma, similar to Examples 4.7 and 4.8. (e) is hard if approached directly. A simpler solution is by contradiction; if $L$ is assumed regular, so is $\overline{L}$. But we already know that, in this particular case, $\overline{L}$ is not regular. I think that all parts of this exercise should be done before proceeding to harder problems.

**5:** This set is harder than Exercise 4, since some algebra is required. For example, in (a), pick string $a^n$. Assume that the string to be pumped is of length $k$, so that by pumping $i$ times we get a string of length $n+(i-1)\,k$. By picking $i = n+1$ we get a string whose length is not a prime number. Part (b) follows from (a), since the language here is the complement of that in (a). Parts (c) and (d) require similar algebraic manipulations. (e) is similar to (a), but (f) is a trick problem since the language is just $\{a\}^*$. There are always a few students who manage to apply the pumping lemma to "prove" that the language is not regular.

**6:** A straightforward application of the pumping lemma.

**7:** Here it is hard to use the pumping lemma directly; instead note that $\overline{L} \cap L(a^*b^*) = \{a^n b^k : n = k-1\}$. We can easily pump out this language.

**8:** False. Take $L_1 = \{a^n b^m : n \geq m\}$ and $L_2 = \{a^n b^m : n < m\}$. Both these languages are nonregular, but $L_1 \cup L_2$ is regular. Many students have difficulty in getting a handle on this problem and try in vain to apply the pumping lemma.

**9:** Good exercise for students to develop some intuition about what is and what is not a regular language. The key is how much has to be remembered by the automaton. Answers: (a) regular, (b) not regular, (c) not regular, (d) not regular, (e) not regular, (f) regular, (g) not regular.

**10:** No, but it is hard to apply the pumping lemma directly. Instead, look at $\overline{L} \cap L\left((a+b)^* c (a+b)^*\right) = \{w_1 c w_2 : w_1 = w_2\}$. The latter is clearly not regular.

**11:** Yes, follows from known closure properties since $L = L_1 \cap L_2^R$.

**12:** Take $n+1$ different strings of the same length and such that $w_i \neq w_j^R$.

**13:** Somewhat difficult, but with a good contrast between (a) and (b).

**14:** The answer is no, but this is difficult to prove. It seems impossible to pick a string to which Theorem 4.8 can be applied successfully. Suppose we pick

$$w = a^m b^m b^m a^{m+1}.$$

Then the adversary simply chooses $x = \lambda$ and $y = a$. We cannot win now by any choice of $i$. We can do better by applying the extended version of the pumping lemma as stated in Exercises 1 and 2 previously. We now choose

$$w = abbba^m b^m b^m \boxed{a^m} bbbaa,$$

and force the adversary to pick a decomposition in the boxed part. The result can be pumped out of the language, although it takes a fair amount of arguing to convince yourself that this is so.

**15:** A classical example: take $L_i = \{a^i b^i\}$. Each language is finite and therefore regular. Their union is the nonregular language $\{a^n b^n : n \geq 0\}$.

**16:** A theoretical exercise, filling a gap left in a previous argument. The proof can be done by induction on the number of vertices.

**17:** A very difficult, but solved exercise.

**18:** Very similar in appearance to Exercise 12 in Section 4.1, but with a different conclusion. The fact that $L_1$ is not finite makes a difference. $L_2$ is not necessarily regular, as shown by $L_1 = \Sigma^*, L_2 = \{a^n b^n : n \geq 1\}$.

**19:** Rectangles are described by $u^n r^m d^n l^m$. Apply the pumping lemma to show that the language is not regular.

# Chapter 5

## Section 5.1

The material in this section is important, but not hard to understand. Most of the exercises explore the concepts in a fairly direct way. Nevertheless, an occasional problem can be difficult. Many of the exercises are reminiscent of (or extensions to) the grammar problems in Section 1.2.

**1:** Straightforward reasoning: first, $S \Rightarrow abB$. By induction it follows easily that $B \overset{*}{\Rightarrow} (bbaa)^n B (ba)^n$. Finally, $B \Rightarrow bbAa \Rightarrow bba$, so that $S \overset{*}{\Rightarrow} ab (bbaa)^n bba (ba)^n$.

**2 and 3:** Simple drill exercise.

**4:** Similar to Example 1.13. Clearly, every sentential form has $n_a = n_b$. Then, by induction on the number of steps in the derivation, we can show that $n_a(v) \leq n_b(v)$. This shows that every string generated is of the stated form. As in Example 1.13, we must also show that every such string can be generated.

**5:** No, by an easy application of the pumping lemma for regular languages.

**6:** Fill-in-the-details via an induction on the number of steps in the derivation.

**7:** Most of the exercises in this set should not prove too difficult for students who have previously encountered such questions in Section 1.2. The problems increase somewhat in difficulty from (a) to (f). Several of the exercises simplify when split into subsets (e.g., $n \neq m$ can be divided into $n < m$ and $n > m$). Answers:

(a) $S \to aSb \,|A|\, B, A \to \lambda \,|a|\, aa \,|aaa, B \to bB|\, b$

(b) $S \to aSb \,|A|\, B, A \to aA \,|\lambda, B \to bbC, C \to bC|\, \lambda$

(c) $S \to aaSb \,|A|\, B, A \to aA \,|a, B \to bB|\, b$

(d) $S \to aSbb \,|aSbbb|\, \lambda$

(e), (f), and (g) follow the arguments in Exercise 16, Section 1.2.

**8:** A set similar in difficulty to Exercise 7. Again, they are much easier to see if split intelligently. For example, in (a) we let $L_1 = \{a^n b^m c^k : n = m\}$ and $L_2 = \{a^n b^m c^k : m \leq k\}$. Then we use $S \to S_1 | S_2$, where $S_1$ derives $L_1$ and $S_2$ derives $L_2$. The other parts are similar in nature.

**9:** Easy, once you see that $head(L) = L(a^* b^*)$.

**10:** An easy exercise, involving two familiar ideas. An answer:

$$S \to aSb \,|S_1, S_1 \to aS_1a|\, bS_1b|\lambda.$$

**11:** Introduce new variables and rewrite rules so that all terminals are on the left side of the productions. For example, $S \to aSb$ becomes $S \to aSB, B \to b$. Then add productions $A \to \lambda$ for all variables that can derive a terminal string. This anticipates some of the grammar manipulations of Chapter 6 and is easier after that material has been covered. At this stage, it can be quite difficult.

**12:** This exercise anticipates the closure results of Chapter 8. The problems are not too difficult, for example, the solution to part (b) can be constructed by $S \to S_1 ... S_1$, where $S_1$ derives $L$. The hardest part of the exercise is to find a grammar for $\overline{L}$ because this first requires a characterization of $\overline{L}$. We can do so by splitting the problem into various cases, such as $\{a^n b^m\}$ with $n > m$, etc.

**13:** Another simple exercise, anticipating closure under union.

**14:** For an answer, use $S \to S_1 S_2$ with $S_1$ deriving $(a+b)(a+b)$ and $S_2 \to aS_2a \,|bS_2b|\, S_1$.

**16:** A little easier than solved Exercise 15.

**17:** A fairly difficult, but instructive problem. The grammar must be forced to generate different symbols in at least one corresponding position. An answer:

$$\begin{aligned} S &\to aSa \,|bSb|\, A \\ A &\to aBb|bAa \\ B &\to aB \,|Ba \,|bB \,|Bb|\, \lambda. \end{aligned}$$

**18:** Routine drill to give the derivation tree, but an intuitive characterization of the underlying language is quite hard to make. One way is to describe the language as

$$L = \{w : 2n_a(w) = n_b(w), w = aubyb, \text{ with } u \in L, y \in L\}.$$

**20:** Simple, but there are lots of different answers.

**21:** $S \to [S] \,|(S)|\, SS|\lambda$.

**22:** This exercise is not hard, but it is worthwhile since it introduces an important idea. You can view this as an example of a metalanguage, that is, a language about languages.

**23:** Same idea as in Exercise 22.

**24:** For a leftmost derivation, traverse tree in preorder, that is, process root, process left subtree, process right subtree, recursively, expanding each variable in turn. The fact that we can do this shows that a leftmost derivation is always possible. For a rightmost derivation, use a traversal defined by: process root, process right subtree, process left subtree.

**25:** Expands on discussion in Example 5.3. For the case $n > m$ we can use the linear productions $S \to aS\,|A, A \to aAb|\,\lambda$, with a similar set for the case $n < m$.

**26:** A fairly simple exercise involving some counting of leaves in a tree. The major purpose is to get students to think about derivation trees. To get the answer, establish a relation between the height of the derivation tree and the maximum number of leaves. For a tree of height $h$, the maximum number is $h^k$. The other extreme is when there is only one node (i.e., there is only one variable in $v$). In this case, each level except the last has $k-1$ leaves so $|w-1| = h\,(k-1)$.

## Section 5.2

**1 to 3:** While finding a grammar for these languages is trivial, the requirement that they be s-grammars makes the exercises a little more challenging.

**4:** Clearly, there is no choice in a leftmost derivation.

**5:** For each variable $A$ on the left, there are at most $|T|$ possible right sides so that $|P| \leq |V|\,|T|$.

**7:** The language is $L\,(aa^*b)$, so the answer is trivial.

**8:** A routine drill exercise.

**9:** Construct a dfa and from it a regular grammar. Since the dfa never has a choice, there will never be a choice in the productions. In fact, the resulting grammar is an s-grammar.

**10:** Straightforward modification of Example 5.12. Follow the approach used for arithmetic expressions.

**11:** Yes, for example $S \to aS_1|ab, S_1 \to b$. Contrast this with Exercise 9 to show the difference between ambiguity in a grammar and ambiguity of a language.

**12:** The grammar $S \to aSa\,|bSb|\,\lambda$ is not an s-grammar, so we cannot immediately claim that it is unambiguous. But, by comparing the sentential form with the string to be parsed, we see that there is never a choice in what production to apply, so the grammar is unambiguous.

**13:** Consider $w = abab$ which has two leftmost derivations

$$S \Rightarrow aSbS \Rightarrow abS \Rightarrow abab$$

and

$$S \Rightarrow aSbS \Rightarrow aSb \Rightarrow abab.$$

**14 and 15:** Simple variations on the current theme.

**16:** The grammar has all s-grammar productions, except for $B \to A$ and $B \to \lambda$, but we can always tell which to use.

**17:** The first part is a somewhat tedious drill. For the second part, note that $A \to bBb$ produces two terminals, and $B \to A$ none. So every two productions produce two terminals and we have at most $|w|$ rounds.

**18:** Simple reasoning. Consider leftmost productions. Since the variable to be expanded occurs on the left of only one production, there is never any choice.

**19:** $S \to aA\,|aaAA, A \to bAb|\,bb$. This exercise anticipates some of the complex substitutions in Chapter 6.

## Section 5.3

A set of rather simple and not very exciting exercises. Some of these are worthwhile as they introduce (or review) BNF and its use in programming language definition. As usual, the main purpose of this type of exercise is to remind students of potential applications. Assign one or two of these if you feel that such a reminder is necessary.

# Chapter 6

Many of the exercises in this chapter are conceptually easy, but tend to be lengthy. This reflects the material of this chapter, whose main difficulties are technical. The exercises are useful to reinforce the constructions. Without some of them, the students may be left with some misunderstandings.

## Section 6.1

**1:** This involves filling in some missing steps, using the given reasoning in reverse.

**2:** A reminder of derivation trees.

**3:** Simple, just substitute for $B$.

**4:** It is not clear how to interpret the algorithm if $A = B$.

**5:** A routine application of the algorithm in Theorem 6.2. The grammar generates the empty set.

**6:** Straightforward, but shows that order matters.

**7 to 9:** Routing applications of the given theorems.

**10 and 11:** Involve elementary arguments to complete some missing detail.

**12:** Since $S$ is a nullable variable, we get

$$S \to ab\,|aSb|\,SS$$

which derives the original language without $\lambda$. This leads into the next exercise, where the particular observation is made general.

**13:** This generalizes the previous exercise. To prove it, show that every nonempty string in $L(G)$ can still be derived.

**14:**

$$S \to aAb|AA$$
$$A \to aAb|\lambda.$$

**15:** We add $A \to y_1\,|y_2|\cdots$. But none of the $y_i$ are empty, so that no $\lambda$-productions can be added.

**16:** Here we add no productions at all.

**17:** Addresses an issue that is pretty obvious to students, but which some find difficult to justify rigorously. An argument, which is at least plausible, can be made from the derivation tree. Since the tree does not embody any order, the order of replacement of variables cannot matter.

**18:** This rather trivial substitution can be justified by a straightforward argument. Show that every string derivable by the first grammar is also derivable by the second, and vice versa.

**19:** An important point: the modified process does not work correctly. Counterexample:

$$S \to abc|AB$$
$$A \to ac.$$

$B$ will be recognized as useless, but not $A$.

**20:** Looks harder than it is. Obviously, in removing useless variables we introduce no new productions; therefore the complexity decreases. For the others, though, one can find simple examples where the complexity is increased.

**21:** Also simple, for example $S \to aA$, $A \to a$ does not have minimal complexity. Serves to point out that just removing useless variables does not simplify the grammar as far as possible.

Exercises 20 and 21 are intimidating in appearance, but really quite simple, once the notation is understood. I include them, because students sometimes ask: "What is the simplest grammar for this language?" so it is useful to point out that "simplest" has to be defined before such a question becomes meaningful.

**22:** A difficult problem because the result is hard to see intuitively. The proof can be done by showing that crucial sentential forms generated by one grammar can also be generated by the other one. The important step in this is to show that if

$$A \overset{*}{\Rightarrow} Ax_k \ldots x_j x_i \Rightarrow y_r x_k \ldots x_j x_i,$$

then with the modified grammar we can make the derivation

$$A \Rightarrow y_r Z \overset{*}{\Rightarrow} y_r x_k \ldots x_j x_i.$$

This is actually an important result, dealing with the removal of certain left-recursive productions from the grammar and is needed if one were to discuss the general algorithm for converting a grammar into Greibach normal form.

**23:** A routine application illustrating the result of Exercise 22.

**24:** The arguments are similar to those in Exercise 22.

## Section 6.2

**1:** Straightforward even though a bit theoretical. By now the students should have no trouble with this sort of thing.

**2 and 3:** Routine drills, involving an easy application of Theorem 6.7.

**4 and 5:** To apply the method described in Theorem 6.7, we need to remove $\lambda$-productions first. The rest is easy, but a little lengthy.

**6:** An exercise that looks harder than it is. It can be solved with elementary arguments. Enumerate the productions generated in each step. In the first, we introduce $V_a$ for all $a \in T$ and $|T|$ new productions $V_a \to a$. In the second, right sides of length $k$ are split into $k-1$ separate rules. The stated result follows easily.

**7:** A trivial exercise, just to get students to work with the widely used concept of a dependency graph.

**8:** A relatively simple exercise leading to a normal form for linear grammars. Consider

$$A \to a_1 a_2 \cdots B b_1 b_2 \cdots b_m.$$

Introduce a new variable $V_1$ with

$$V_1 \to a_2 \cdots B b_1 b_2 \cdots b_m$$

and

$$A \to a_1 V_1.$$

Continue this until you get

$$V_{k-1} \to a_k V_k.$$

Then apply the same idea to remove multiple terminals from the right.

**9:** Another normal form, which can be obtained from Chomsky normal form. Productions $A \to BC$ are permitted. For $A \to a$, create new variables $V_1$, $V_2$ and productions $A \to aV_1V_2$, $V_1 \to \lambda$, $V_2 \to \lambda$.

**10 to 13:** These are relatively simple problems whose answers can be found from first principles. They serve to make it plausible that conversion to Greibach normal form is always possible.

**14:** Yes, by a straightforward substitution.

**15:** This solved problem is not too bad, but its generalization in the next exercise can be quite difficult.

**16:** This exercise shows an important extension of the material in the text. It is for most a very difficult exercise, since it involves a good grasp of the use of substitutions. Start from Greibach normal form, then make substitutions to reduce the number of variables in the productions. We illustrate this step with

$$A \to aBCD.$$

First introduce a new variable $V_1$ and productions

$$A \to aV_1$$

and

$$V_1 \to BCD.$$

The second production is not in correct form so we continue, introducing $V_2$ and

$$V_1 \to BV_2$$

and

$$V_2 \to CD.$$

After a while, all productions will be either in correct form or of the form

$$V_k \to Bv_j.$$

But the rules with $B$ on the left are either

$$B \to b$$

or

$$B \to bV_j.$$

The final substitution then gives the right form. Complete arguments can be found in some advanced treatments, for example, in Harrison 1978. This exercise may be suitable for an extra-credit assignment for the better students.

### Sections 6.3

The material in this section is optional. The CYK algorithm is difficult to grasp and not particularly important for understanding the concepts in this text. Still, it is a good example of dynamic programming and students can benefit from studying it. The exercises in this section can be used to complement what might be a very short discussion in class. Exercises 1 and 2 are drill, requiring no more than an understanding of the notation by which the algorithm is described. Exercises 3 and 4 involve an extension and actual implementation. For this, the student will have to have a good understanding of the construction.

## Chapter 7

### Section 7.1

Here we have a collection of problems of varying difficulty. Particularly instructive are the ones that force the student to think nondeterministically. This is really the first place where nondeterminism becomes an important issue. For finite automata, a deterministic solution can always be constructed, so nondeterminism sometimes appears as a technical trick. Here the situation is different. In problems such as Exercise 4(f), students often start by trying to find a deterministic solution and will arrive at the nondeterministic approach only after a few failures. But once they discover the solution, they begin to understand something about nondeterminism. An even more convincing problem is Exercise 10 in Section 8.1.

**1:** No need for state $q_1$; substitute $q_0$ wherever $q_1$ occurs. The major difficulty is to argue convincingly that this works.

**2:** This problem is not too hard, but requires a little bit of analysis of the nondeterministic nature of the pda in Example 7.4. For this reason, it is a helpful exercise. An argument goes somewhat like this: the switch to state $q_1$ is done nondeterministically, and can be made any time. But if it is not made in the middle of the string, the emergence of the stack start symbol will not coincide with the end of the input. The only way the stack can be cleared is to make the transition in the middle of the input string.

**3:** Because the languages involved are all regular, the npda's can be constructed as nfa's with an inactive stack. One of these problems may be useful in illustrating this point.

**4:** A set of exercises in programming a pda. Most students will have some experience in programming with stacks, so this is not too far removed from their experience and consequently tends to be easy. Those problems, such as (f) and (j), which illustrate nondeterminism may be a little harder. Part (a) is easy: put two tokens on stack for each $a$, remove one for each $b$. Parts (b) and (c) are also easy. Part (d) is a little harder. Put a token on stack for each $a$. Each $b$ will consume one until the stack start symbol appears. At that time, switch state, so now $b$ puts on tokens to be consumed by $c$. In (e), use internal states to count three $a$'s. In (f) nondeterminism is the key, an $a$ can put one, two, or three tokens on the stack. Parts (g), (h), (i) are easy, following the approach suggested in Example 7.3. Part (j) is similar, but uses nondeterminism.

**5:** Students who blindly follow the lead suggested by Example 7.2 will make a number of subtle errors, but the approach is still useful. The best way to proceed is to split the npda initially into cases $n > m$ and $n < m$. The first can be dealt with by putting the pda into a final state with $\delta(q_2, \lambda, 1) = \{(q_f, \lambda)\}$. In the second case, the stack start symbol will appear before all the $b$'s are read, and one must be careful to check the rest of the string (otherwise, something like $abba$ might be accepted).

**6:** A simple problem. Put $w_1$ in stack, then go into final trap state whenever a mismatch with $w_2$ is detected.

**7:** This is a little harder than Exercise 6, since the part belonging to $L(a^*)$ must not be put on stack. We can use nondeterminism to decide where this part ends.

**8:** A lengthy, but not difficult problem. The reason for the lengthiness is that internal states must be used to recognize substrings such as $ab$ as a unit. This substring puts a single symbol on the stack, which is then consumed by the substring $ba$.

**9 and 10:** These are simple exercises that require an analysis of a given pda.

**11:** A simple tracing exercise.

**12:** Any string will be accepted since there are no dead configurations.

**13:** Adds $\lambda$ to the original language.

**14:** See solved exercises

**15:** No change. Since the original machine accepts $w$, we have

$$(q_0, w, z) \overset{*}{\vdash} (q_2, \lambda, 0) \vdash (q_3, \lambda, \lambda).$$

But in the modified machine

$$(q_0, w, z) \overset{*}{\vdash} (q_2, \lambda, 0) \vdash (q_0, \lambda, 0) \vdash (q_3, \lambda, \lambda).$$

**16:** Not a very difficult construction, but worthwhile. The trick is to remember the extra symbols using internal states, for example,

$$\delta(q_i, a, b) = \{(q_j, cde)\}$$

is replaced by

$$\delta(q_i, a, b) = \{(q_{jc}, de)\}$$
$$\delta(q_{jc}, \lambda, d) = \{(q_j, cd)\}.$$

This result is needed in subsequent discussions.

**17:** In many books this is treated as a major issue. We left it out of our main discussion, but the result is useful. The construction is not hard to discover. Whenever the pda goes into a final state, it can be put into a stack-clearing mode. Conversely, if $\widehat{M}$ sees an empty stack, it can be put into a final state. It's a good exercise to have students make the argument precise. This result is also needed later.

## Section 7.2

There are some difficult constructions in this section, particularly in Theorem 7.2. A few exercises will make the results plausible, which is all one really needs.

**1:** Straightforward drill exercise.

**2:** A relatively easy proof that just formalizes some obvious observations.

**3:** The long way is to follow the construction in Theorem 7.1. The short way is to notice that

$$L(G) = \{a^{n+2}b^{2n+1} : n \geq 0\}.$$

**4 to 6:** The grammars should be converted to Greibach normal form, after which the exercises are direct applications of the construction of Theorem 7.1.

**7:** This exercise requires that the student be able to put together two observations made in the text. In Theorem 7.1 we constructed a three-state pda for any context-free grammar. From Theorem 7.2 we get a context-free grammar for every npda. Putting the two together proves the claim.

**8:** This may be quite hard to see and involves a careful examination of the proof of Theorem 7.1. The state $q_1$ serves the sole purpose of assuring that the desired derivation gets started. If we replace $q_1$ by $q_0$ everywhere, we change very little, except that Equation (7.3) now becomes

$$\delta(q_0, \lambda, z) = \{(q_f, z)\}$$

and $\lambda$ is also accepted. To fix this, introduce another special symbol, say $z_1$, then use

$$\delta(q_0, \lambda, z) = \{(q_0, Sz_1)\}$$

and

$$\delta(q_0, \lambda, z_1) = \{(q_f, \lambda)\}.$$

**9 and 10:** Two similar exercises, one of which is solved.

**11 and 12:** These two exercises illustrate Theorem 7.2. Since the proof of the theorem is tedious and intricate, you may want to skip it altogether. In that case, these two exercises can be used to make the result more believable.

**13:** May be worthwhile, since it expands on a comment made in the proof of Theorem 7.2.

**14 and 15:** Routine use of a given construction.

**16:** Explores a point treated somewhat briefly in the text. Two previous exercises (Exercises 16 and 17 in Section 7.1) are similar to what is needed here and give an idea for a construction that can be used.

**17:** This fill-in-the-details exercise is long and tedious and is worthwhile only if you are concerned with tying up all loose ends.

**18:** This is a worthwhile exercise in algorithm construction. The basic process is sketched on p. 185, so most students should be able to provide the details.

## Section 7.3

**1 to 3:** Easy modifications of Example 7.8.

**4:** A little harder, with a given solution.

**5:** The pda is not deterministic because $\delta(q_0, \lambda, z)$ and $\delta(q_0, a, z)$ violate the conditions of Definition 7.3. For a deterministic solution, make $q_f$ the initial state and use $\delta(q_f, a, z) = \{(q_0, 0z)\}$, $\delta(q_f, b, z) = \{(q_0, 1z)\}$. The $\lambda$-transition can then be omitted, with everything else remaining.

**6:** This is fairly easy to see. When the stack is empty, start the whole process again.

**7:** Somewhat vague, but worthwhile in anticipation of results in Section 8.1. Intuitively, we can check $n = m$ or $m = k$ if we know which case we want at the beginning of the string. But we have no way of deciding this deterministically. I like to assign this kind of exercise because it encourages students to develop some intuitive insight, but students tend to find them a little frustrating because it is hard to say exactly what is expected. The answers are also hard to grade.

**8:** The language is deterministic. Construct the pda as you would for $n = m + 2$. When all $b$'s have been matched, check if there are two more $b$'s.

**9:** A straightforward exercise. The $c$ makes it obvious where the middle of the string is.

**10:** This makes a good contrast to Exercise 9. Since there is no marker, we need to guess where the middle is. It seems plausible that this cannot be done deterministically, but the proof is very difficult and you cannot expect more than just some hand-waving motivation.

**11:** Proceed as in the case $n_a(w) = n_b(w)$, Example 7.3 and Exercise 5 above. When $z$ comes to the top of the stack and we are not in state $q_f$, accept the string.

**12 to 14:** Here the student is asked to work out some details omitted in the text. The arguments should not be too hard. For example, suppose $a^n b^n c^k$ were accepted, making the $\lambda$-transition shown in Figure 7.2. By not making the $\lambda$-transition, we could accept $a^n b^{n+k}$, which contradicts what we know about $M$.

**15:** A dfa can be considered a dpda whose stack is irrelevant. This intuitive reasoning is easy, but if you ask for more detail, the argument tends to become lengthy.

Exercises 16 and 17 anticipate later results, and both have some nonobvious constructions that can be made along the lines of the intersection construction in Theorem 4.1. If $Q$ and $P$ are the state sets for the dfa accepting $L_2$ and the dpa accepting $L_1$, respectively, the control unit has states $Q \times P$. The stack is handled as for $L_1$. The key is that $L_2$ is regular, so that the combined machine needs only a single stack and is therefore a pda.

**18:** It is hard to discover an example of this, but a modification of Example 7.9 will do. Let $L_1 = \{a^n b^n : n \geq 0\}$ and $L_2 = \{a^n b^{2n} c : n \geq 0\}$. Then clearly $L_1 \cup L_2$ is nondeterministic. But the reverse of this is $\{b^n a^n\} \cup \{cb^{2n} a^n\}$, which can be recognized deterministically by looking at the first symbol.

## Section 7.4

The material in this section is important in a course on compilers, but dispensable here and therefore treated only very briefly. A few exercises here, though, are convincing as applications and point the way to more complicated issues treated in compilers courses. Constructing $LL$ grammars, and proving that they are indeed $LL$, tends to be quite difficult.

**1:** A reasonably simple argument to show that the grammar is $LL(3)$. If the next two symbols are $ab$ and the next is not the end of the string, apply $S \to abS$, otherwise use $S \to aSbS$. We can be more formal and use Definition 7.5, but you will have to append some sort of end-of-string symbol to make it come out exactly right.

**3:** In my observation, this is nearly impossible for a student who approaches the problem from first principles. A more workable, but lengthy approach first constructs a dpda for the language (easy), then uses the construction of Theorem 7.2.

**5:** By definition, at most one production can be applied at any step, so there is a unique leftmost derivation.

**6 and 7:** An exploration of some connections that were not explicitly mentioned in the text. The arguments can be made using the established connection between pda's and grammars, where in the pda a finite look-ahead can be built into the control unit. Formal arguments are involved, making this an extra-credit problem for the very good students.

8: Obviously, if the grammar is in GNF, we can see by inspection if it is $LL(1)$. To see if it is $LL(2)$, substitute for the leftmost variable. For example

$$A \to aB$$
$$A \to aC$$
$$B \to bD$$
$$C \to cE$$

is not $LL(1)$. But if we substitute to get

$$A \to abD$$
$$A \to acE$$

then by looking at two symbols, we know how to expand $A$ and we can tell if the grammar is $LL(2)$.

9: (a) is easy, but the exercises get progressively harder. (d) and (e) are very hard to do from first principles, but are more manageable if you construct a dpda first.

# Chapter 8

## Section 8.1

Having worked with pumping lemmas for regular languages, students can be expected to be familiar with the basic idea behind such arguments. What is new here is that the opponent has many more choices, all of which must be addressed if the argument is to be complete. The most common mistake is that students forget to take everything into account.

Exercises 1 to 7 all involve straightforward application of the pumping lemma and should not be hard for those who have a good grasp of it. Nevertheless, many students will get confused because of the inherent difficulty of the pumping lemma.

8: This set of examples is a little more interesting than the previous ones, since we are not told to which language family the examples belong. It is important that the students develop some intuitive feeling for this. Before a correct argument can be made, a good conjecture is necessary. Answers: (a) context-free, (b) not context-free, (c) context-free, and a good contrast with (b), (d) context-free, (e) not context-free, (f) not context-free. Note the different conclusions of (b) and (c).

**9:** A reasonably easy fill-in-the-details exercise. The string must be long enough so that some variable repeats in the leftmost position. This requires $|V|$ steps. If the length of the right side of any productions is not more than $N$, then in any string of length $|V|\,N$ or longer, such a repetition must occur.

**10:** Difficult, with a surprising answer.

**11:** An easy exercise. Show that it is context-free by giving a context-free grammar. Then apply the pumping lemma for linear languages to show that it is not linear.

**12:** Choose as the string to be pumped $a^m b^{2m} a^m$.

**13:** A two-part problem, illustrating existence and non-existence arguments.

**14:** The language is linear. A grammar for it is

$$S \to aSb \,|aaSb|\, ab.$$

**15:** The language is not linear, although not every string can be pumped out of the language. A string that does work is

$$w = (\ldots(a)\ldots) + (\ldots(a)\ldots)$$

where $(\ldots($ and $)\ldots)$ stand for $m$ parentheses.

**16:** Effectively the same reasoning as Exercise 9 above.

**17:** A fairly easy exercise that involves an examination of the procedures for removing $\lambda$-productions and unit-productions. It is straightforward to argue that if we start with a linear grammar, the constructions do not introduce any nonlinear productions.

**18:** This is not a context-free language. If we choose as our string something like $a = 111\ldots10, b = a + 1$, then the adversary has few choices, all of which are easily pumped out of the language.

**19:** A difficult problem; considerably harder than Exercise 6. It takes much more thought to find a string that can be pumped out of the language. One answer: Choose $n = (m!)^2 + m!$ If the adversary picks a string of length $r \leq m$ to be pumped, we choose $i$ such that

$$(m!)^2 + m! + ir = (m! + r)^2$$

$$i = \frac{m!\,(2r-1) + r^2}{r}.$$

Since $m!/r$ is an integer, it is always possible to choose an integer $i$ to pump the string out of the language.

**20:** Similar to and not much harder than Exercise 2. Choose $a^{pq}$, with $p, q < m$, then pump the chosen substring $pq$ times.

## Section 8.2

**2:** An easy problem with a simple answer

$$S \to Sc|A|\lambda$$
$$A \to aAb|\lambda.$$

Exercises 3 to 5 are theoretical, but easy. Students should have no trouble with such elementary arguments.

**3:** Consider all productions. For each $a \in T$, substitute $h(a)$. The new set of productions gives a context-free grammar $\widehat{G}$. A simple induction shows that $L\left(\widehat{G}\right) = h(L)$.

**4:** Same reasoning as in Exercise 3.

**5:** Given $G$, construct $\widehat{G}$ by replacing $A \to x$ with $A \to x^R$. Then prove by induction on the number of steps in the derivation that if $x$ is a sentential form of $G$, then $x^R$ is a sentential form of $\widehat{G}$.

**6:** Closed under reversal: regular, linear, and context-free languages. Not closed under reversal: deterministic context-free languages. The last observation comes out of Exercise 18, Section 7.3.

**7:** A little challenging, but the arguments are similar to those in Theorem 8.5. Consider $\Sigma^* - L = \overline{L}$. If context-free languages were closed under difference, then $\overline{L}$ would always be context-free, in contradiction to previously established results.

To show that the closure result holds for $L_2$ regular, do the cross-product construction suggested in Theorem 8.5, accepting a string if it ends in a state belonging to $F_1 \times \overline{F}_2$.

**8:** Same arguments as in Exercise 7. The construction does not introduce any nondeterminism if the pda for $L_1$ is deterministic.

**9:** For union, combine grammars. This does not affect linearity. For non-closure under concatenation, take $L = \{a^n b^n\}$. Then as we can show by a simple application of the pumping lemma, $L^2$ is not linear.

**10:** There is a simple answer, but it is easy for students to get off on the wrong track. The languages $L_1$ and $L_2$ in Theorem 8.4 are linear, but their intersection is not context-free, so it is not linear.

**11:** That deterministic context-free languages are not closed under union was shown in Example 7.9. Intersection follows from the languages in Theorem 8.4.

**12:** This problem requires a chain of reasoning. The necessary pieces are known, so it is just a matter of finding what is needed. The language $L$ in Exercise 10, Section 8.1 is context-free, but its complement is not. If it were, then the language $\overline{L} \cap L(a^*b^*)$ would be context-free by Exercise 7. But this is the non-context-free language $\{wcw\}$. This is a worthwhile but quite difficult problem unless students have done (and remember) Exercise 10 in Section 8.1.

**13:** Not an easy problem. The trick is first to use a left-linear grammar for $L_2$. In this grammar, every production that leads immediately to a terminal string, say, $A \to ab...$, is replaced by $A \to S_1ab...$, where $S_1$ is the start symbol for a linear grammar that derives $L_1$. We then derive $S \overset{*}{\Rightarrow} Aw_2 \Rightarrow S_1ab...w_2$, where $ab...w_2 \in L_2$. After $S_1$ is used to derive a string in $L_1$, the result is in $L_1L_2$.

**14:** This is an easy problem if you take Example 5.13 as given. The languages $L_1 = \{a^nb^nc^m\}$ and $L_2 = \{a^nb^mc^m\}$ are both unambiguous, but, as we claimed in Example 5.13, their union is not.

**15:** The intersection of the two languages in the solution to Exercise 14 is not even context-free.

**16:** $L_1$ is not necessarily deterministic. As counterexample, take $L = \{a^nb^n\} \cup \{ca^nb^{2n}\}$.

**17:** One can construct a context-free grammar for this language, but a simpler argument is to use Theorem 8.5. $L$ is the intersection of the context-free language $\{a^nb^n : n \geq 0\}$ and the regular language $\{a^kb^m : k$ is not a multiple of $5\}$.

**18:** Can be solved by the same approach as Exercise 17.

**19:** Yes. The argument requires an easy, but lengthy construction. Consider a dpda for such a language. Modify it so that whenever it "consumes" a terminal $a$, the modified pda consumes $h(a)$. The result of the construction is still deterministic.

**20:** Fill-in-the-details as practice for induction.

**21:** Use the algorithm for finding nullable variables in Theorem 6.4.

**22:** Rewrite the grammar in Greibach normal form. Then enumerate all partial derivations of length $n$ or less to see if any of them yield a sentence.

**23:** Use Theorem 8.5, together with Theorem 8.6.

# Chapter 9

In the discussion of Turing machines in Chapter 9 and 10, I have changed the manner of presentation to some extent, abandoning the theorem-proof style of discussion in favor of more descriptive arguments. I believe this to be necessary. Turing machines are too complex to be treated in full detail. We cannot talk about them at the same level of detail as we did for, say, finite automata without running into very lengthy and boring arguments that obscure essential points. What I do here is to introduce the general ideas (which are easy to understand), then present a number of motivating examples to give the student some confidence that, in principle, everything can be done precisely. Most students find this convincing; the exercises in this chapter are intended to reinforce this conviction.

## Section 9.1

The notion of a Turing machine is not a difficult one and programming Turing machines presents little difficulty to students who are often used to much more challenging programming exercises. What is hard to deal with is the tediousness of the job, and that it is very easy to make mistakes. The exercises in this section are almost all conceptually simple, but many have very lengthy solutions. They are useful for getting a better understanding and to convince the students of the generality of the concept. The second issue, which is the more important one, is not always so easily seen.

If students are asked to do any detailed programming of Turing machines, it is essential that they can check their answers easily. The only way of doing this is to have a Turing machine simulator. Exercise 1 asks that the students write one, but if you elect not to do this, you should provide one for general use.

**1:** This is a programming exercise, but has some good aspects. One is that once written, the program can be used to check the solutions to other problems. Tracing Turing machine programs by hand is not pleasant, so this is useful. The second point is that the program can be viewed as an implementation of a universal Turing machine, an issue that is important in later discussions.

**2:** A three-state solution is

$$\delta(q_0, a) = (q_1, a, R)$$
$$\delta(q_1, a) = \delta(q_1, b) = (q_1, a, R)$$
$$\delta(q_1, \square) = (q_2, \square, R)$$

with $F = \{q_2\}$. With our definition of the language accepted by a Turing machine, it is possible to solve this with two states. Since the only requirement is that the first symbol of the input be an $a$, a two-state solution is

$$\delta(q_0, a) = (q_1, a, R)$$

with $F = \{q_1\}$. The machine reads the first symbol and stops either in a final or nonfinal state, depending on whether or not the symbol read was an $a$.

**3:** A simple exercise in tracing the computation of a Turing machine. The exercise complements the discussion in Example 9.7.

**4:** No.

**5:** A simple problem, with answer $L = L(ab^* + bb^*a)$.

**6:** The Turing machine halts whenever a symbol not in $\{1, \square\}$ is encountered.

**7:** All parts of the exercise are conceptually easy, but some of them are rather tedious. Having a Turing machine simulator will be very helpful.

**8:** A very tedious exercise, which can be solved by finding the middle of the string first, then comparing the two halves. I would accept an outline of the solution, since the details are too messy. The exercise does show, however, that a Turing machine can do something a pda cannot do.

**9 to 11:** These are lengthy, but straightforward exercises.

**13:** Good practice in providing some missing detail.

**14:** Routine exercise.

**15 and 16:** Just for filling in some missing detail.

**17 and 18:** Easy, but useful exercises. They point out that a Turing machine transducer is not tied to unary notation, but can handle more common number representations. The only thing that happens is that the resulting machines tend to become large.

**20:** The empty string is normally ignored in the discussion of the Turing machines. We could include it by changing Definition 9.2 to require that $\lambda \in L(M)$ if and only if

$$\delta(q_0, \square) = (q_f, \square, R).$$

Note that the empty string cannot be included by simply requiring that $q_0 \in F$, since the machine would then accept any string.

### Section 9.2

In this section we explore the generality of the Turing machine and the exercises revolve around this exploration. Conceptually, the exercises present no difficulty, since programming in pseudo-code is nothing new for most students. The trouble with all of this is the usual one when working with pseudo-code: what statements are permissible and how much detail do you expect. I like this kind of exercise and usually do a few of them. But expect the results to be hard to grade and some complaints from students who feel that they don't know what you expect.

### Section 9.3

**1:** Worth thinking about, but hardly something to do in detail.

**2:** A subtle point: Turing machines as defined are deterministic, while the definition of a pda permits nondeterminism. This will be resolved in the next chapter.

**3:** Not a standard exercise, but highly recommended. The article mentioned is a good one and will increase students' confidence in working with Turing machines. A review will make students synthesize and put material in their own perspective. It also serves as a preview to later material.

## Chapter 10

In this chapter we explore variations of the definition of a Turing machine. As in Chapter 9, the discussion is more intuitive and discursive than rigorous and complete. The exercises reflect this orientation; their main purpose is to

convince the student, by a consideration of several alternatives, that Turing machines are indeed universal calculators.

## Section 10.1

Many authors define the standard Turing machine model somewhat differently. These exercises explore some common variations. The answers to most of the exercises are easy to see and, if you accept solutions in broad outline, not hard to answer. The important point here is the concept of simulation, which is explored here in simple settings. Simulation is central to the rest of the discussion in the text.

While even the simple simulations are too lengthy to do in full detail, one aspect that can and should be made precise are definitions of various alternative models. Most of the exercises involve this aspect.

**1 to 4:** These are all easy exercises, involving a formal definition and some simple simulations.

**5:** Here we have a Turing machine with a more complicated transition function. The simulation is simple enough to describe in full detail.

**6 and 7:** In some arguments it is helpful to be able to claim that a Turing machine cannot create or destroy blanks. Such a restriction is not essential since we can always introduce a new "pseudo-blank" symbol in the tape alphabet. The simulation of a standard machine is straightforward.

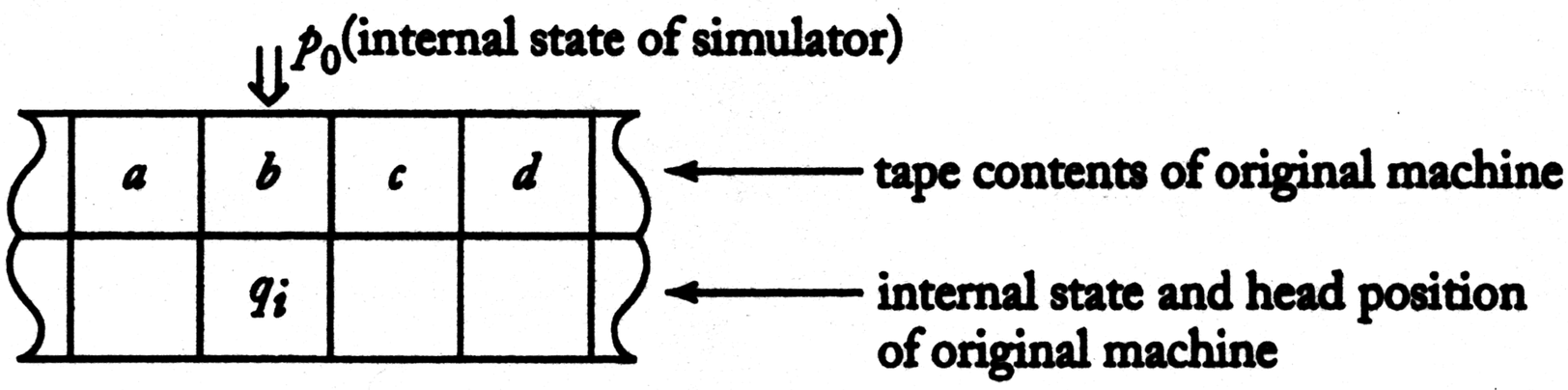
**8:** Another variation that is sometimes convenient. The simulation is done by putting any non-final halting state into an infinite loop. The simulator then can be made to accept any language accepted by a standard Turing machine.

**9 to 11:** Here are some unusual variations whose simulation is straightforward.

## Section 10.2

**1 to 3:** These are exercises in producing formal definitions for some Turing machine variants. Several slightly different definitions are possible, but they all capture the essence. The simulations are not hard. One or two of these problems are worthwhile for exploring different Turing machine models and their equivalence.

**5:** A difficult exercise, as it involves inventing a trick. Most students will see that the simulating machine can keep track of the internal state of the original machine by putting the state number on a separate track, something like



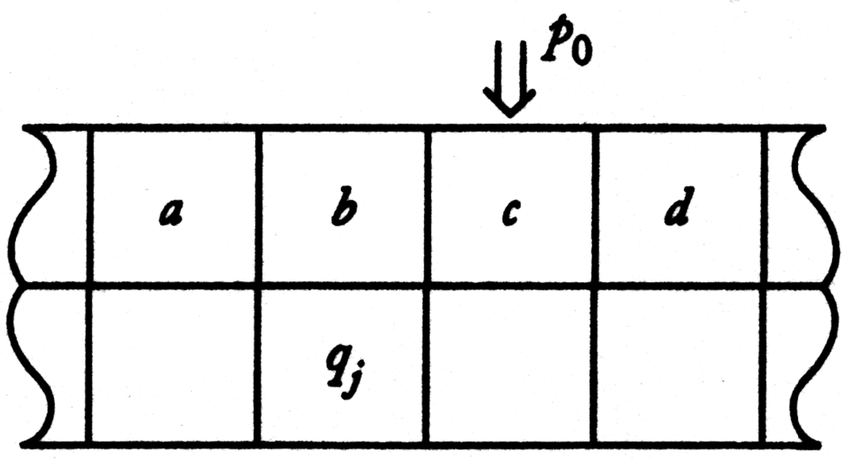

The difficulty comes in simulating moves, say

$$\delta(q_i, b) = (q_j, e, R).$$

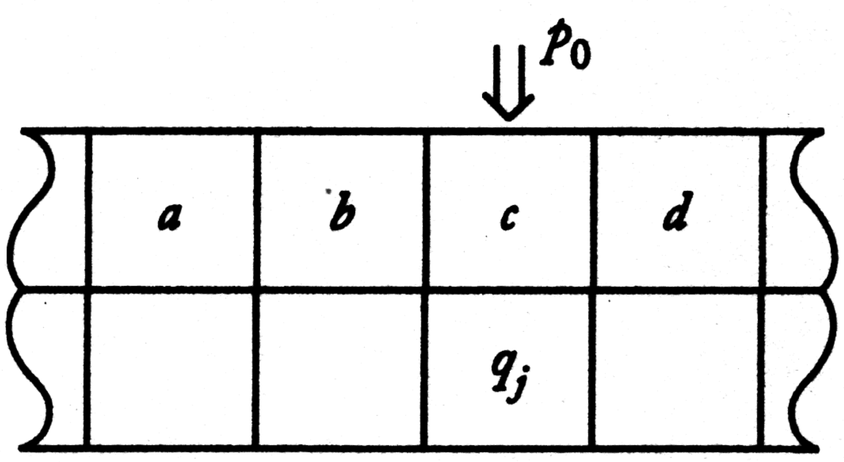
If the simulating machine uses just

$$\delta(p_0, (b, q_i)) = (p_0, (e, q_j), R)$$

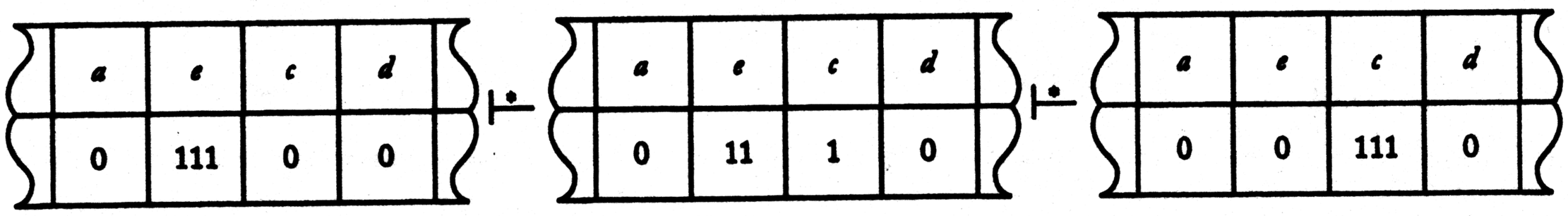
then you end up with




instead of what we want:



To get the $q_j$ under the $c$, we use the following trick. Represent $q_i$ by $i$, then transfer state information in steps as shown below:




Six states make it easy to remember what is going on (transferring $q_i$ to left, transferring to right, etc.).

**6:** Once you see the solution to Exercise 5, this one is not so bad. By using other tracks to remember what is going on, we can reduce the number of required states. The details of the process are of course quite long.

**7:** A difficult problem, suitable only as an extra-credit, research-type of assignment. For a solution, see Hopcroft and Ullman 1979.

**8:** A fairly easy problem; the extra tape makes the simulation much easier than the one in Exercise 5.

**9:** This exercise makes a point that is relevant in a subsequent discussion of complexity.

## Section 10.3

**1 and 2:** Easy conceptually, but with lots of detail.

**3:** The point of this exercise is that a nondeterministic solution is much easier. We guess the middle of the input nondeterministically, then compare the two parts.

**4:** Nondeterministically choose the boundary between $w$ and $w^R$, then match three symbols in each step. A deterministic solution is not much more difficult to visualize–just divide the input into three equal parts.

**5:** The same point as Exercise 4. To solve this deterministically, we need to try various values of $|x|$ and $|y|$; this creates a nontrivial bookkeeping chore.

**6:** Nondeterministically, guess a divisor $a$, then perform division. If there is no remainder, accept the string.

**7:** The most interesting exercise in this section. It makes the point that two stacks are better than one, but three are not better than two and thereby answers a question raised in the preamble to Chapter 9.

## Section 10.4

Most of the exercises in this section explore the notion of proper ordering for various countable sets. The exercises should pose no serious difficulty for anyone who understands the basic idea. The result that fractions can be put into one-to-one correspondence with the integers is a little counter-intuitive and may warrant some exploration. Exercises 7 to 9 serve this purpose.

## Section 10.5

**1:** It is not hard to give details.

**2:** First, divide the input by two. The space released can then be used to store subsequent divisors.

**3:** Since $n$ cells can be in at most $|\Sigma|^n$ states, all of the tape contents can be remembered by the control unit.

**4:** A set of relatively simple, but tedious exercises. If you accept outline solutions, these should not cause any trouble. The full details, on the other hand, are probably not worthwhile. The point to be made is that all examples of this kind, even very complicated ones, can be done with lba's.

**5:** A full-scale solution would be very lengthy and is certainly not worthwhile, but students can build on the discussion in Example 10.5. The key is that the algorithm for this example can be made deterministic so that it terminates for all inputs. When a halting configuration is reached, we simply "complement" the acceptance, which gives an lba for the complement of the original language.

**6 and 7:** These exercises close a gap in our arguments, showing that a pda is less powerful than an lba. The argument can be difficult. The easiest way is an indirect approach, using several established results. We start with a context-free grammar, then use Exercise 15, Section 6.2, to argue that there exists an equivalent grammar in two-standard form. If we then repeat the construction of Theorem 7.1 with this two-standard grammar, we get a pda in which each move consumes an input symbol and never increases the stack content by more than one symbol.

**8:** We can find deterministic solutions for all cases. This exercise hints at the question "Are deterministic lba's equivalent to nondeterministic ones?" This is an open problem.

# Chapter 11

Chapters 9 and 10 contain material that, although technical and often tedious, is easily visualized and therefore presents few conceptual difficulties. In Chapters 11 and 12, the concepts become more difficult. Working with uncountable sets and diagonalization is hard for many students who often have only a very superficial grasp of the arguments. While most of the exercises in this chapter have short answers, many students will find them difficult. The underlying ideas are not very concrete and some students may not even know where to start. The subject matter is often considered graduate material, so it is not always easy for undergraduates.

## Section 11.1

**1:** This is the classical problem of Cantor. We represent real numbers as decimal expansion, then diagonalize this representation.

**3:** First list all elements $L$, then all elements of $L^2$, etc.

**4:** Use the pattern in Figure 11.1 to list $w_{ij}$, where $w_{ij}$ is the $i$-th word in $L^j$. This works because each $L^j$ is enumerable. Contrast this solution with that of Exercise 3.

**5:** If $\overline{L}$ were recursive, then $\overline{(\overline{L})}$ would also be recursive, and therefore recursively enumerable.

**7:** Yes. Use $M_1$ and $M_2$ as in the solution of Exercise 6. Let $M_1$ execute one move, then $M_2$ one move. After that, we let $M_1$ execute its second move, followed by the second move of $M_2$, and so on. If one machine accepts the input, the other can proceed normally; if one machine reaches a configuration indicating that the input is not accepted, the computation can be terminated. The resulting machine accepts any $w \in L(M_1) \cap L(M_2)$.

**8:** The same idea as in Exercises 6 and 7, except that the resulting machine always halts.

**9:** Start by reversing input $w$, then run the Turing machine with input $w^R$.

**10:** Yes. Nondeterministically split $w$ into $w_1 w_2$ such that $w_1 \in L_1$ and $w_2 \in L_2$. The Turing machine for $L_1$ will halt when given $w_1$; then let $M_2$ operate on $w_2$.

**12:** Observe that $L_2 - L_1 = L_2 \cap \overline{L_1}$. But $L_2$ and $\overline{L_1}$ are recursively enumerable, so that the result follows from Exercise 7. As an extension of this exercise, consider $L_1 - L_2$ and show that the result is not necessarily recursive enumerable.

**13:** Since proper order means increasing length, we let the Turing machine generate all strings of length up to $|w|$. If we find no match, we can stop and claim that $w$ is not in the language.

**15:** This looks like a rather vague problem, but one can get a decent answer. It demonstrates that, although $\overline{L}$ is not recursively enumerable, we can still find elements of it. For example, suppose we choose an ordering in which $M_2$ is defined by

$$\delta(q_0, a) = (q_0, a, R).$$

Clearly, $L(M_2) = \varnothing$, so that $a^2 \notin L(M_2)$. Therefore, $a^2 \in \overline{L}$.

**16 and 17:** These are good exercises for exploring countable and uncountable sets. Both proofs are fairly easy if a contradiction approach is used.

**19:** Simple, but good exercise in reasoning with countable and uncountable sets. If the set of all irrationals were countable, then its union with the rationals (i.e. all reals) would also be countable.

## Section 11.2

**2:** It is not easy to see how to approach the problem. The assumption was made mainly to keep the notion of substituting one string for another simple, but it is not really necessary. Perhaps the better students will see that one can always introduce a "pseudo-lambda" $\Lambda$ and generate a new grammar in which every symbol $x$ of $V$ and $T$ in every production is replaced with $\Lambda x \Lambda$.

**4:** The main point of the argument is that although we can generate variables such as $V_{bb}$ and from it derive $V_{b0b}$, no terminal string can be derived from the latter. This is because no $b$ occurs in the transitions (11.8) and (11.9).

**5:** Fill in the details.

**6:** Straightforward, but tedious. A drill exercise to test understanding of the construction of Theorem 11.7.

8: Another normal form. When necessary, introduce new variables. Example

$$ABC \to XYZ$$

can be replaced with

$$BC \to D$$
$$AD \to XW$$
$$W \to YZ.$$

The equivalence argument is not hard.

9: A simple construction once you get the idea. Whenever a production has only terminals on the left, say

$$a \to x,$$

we introduce a new variable $V_a$ and replace every $a$ in $P$ by this variable. To get rid of it eventually, we add $V_a \to a$ to the productions.

This exercise illustrates than in unrestricted grammars the distinction between variables and terminals is blurred. Terminals can be changed, although the final sentence must be composed entirely of terminals.

## Section 11.3

Finding context-sensitive grammars for languages that are not context-free is almost always difficult. The answers are not easy to see, and particular grammars hard to verify. Only in a few instances can reasonably concise plausible results be obtained from first principles. In many cases, students will find it easier to construct an lba first, then obtain the corresponding grammar by the construction in Theorem 11.9. The languages in Exercises 1 and 2 are manageable, but don't expect them to be easy. A variation on these exercises is to give the answer, asking students to give convincing reasons why the grammar works. Exercise 19 in Section 1.2 is suitable here also.

1: (a) and (b) are easiest, as we can follow the pattern established in Example 11.2. The solved part (c) can be used as additional guide.

(d) We give only an outline. We can start with

$$S \to aXaY|bXbY.$$

The variable $X$ then creates some new symbol and a messenger to the right, say by

$$aX \to aaX_a.$$

When $X_a$ meets $y$, it creates an $a$ and another messenger to the left. The details, although lengthy, follow along the lines of part (c).

**2:** The answers tend to be quite long, but the solution can be made easier by using variables subscripted with the terminal they are to create. For example, $V_{ab}$ needs to create an $a$ and a $b$ somewhere. For part (a), we use the start variable $V_{abc}$, and introduce productions like

$$V_{abc} \to abc\,|bca|\cdots|V_{abc}V_{abc}$$
$$V_{ab} \to ab\,|aV_b|\,bV_a|\cdots.$$

This keeps track of the right number of symbols of each kind, but does not take care of the requirement that the symbols can be created anywhere in the string. For this latter requirement, we allow interchanging of variable subscripts, for example

$$V_{bc}V_{abc} \to V_{abc}V_{bc}$$

with similar rules for all other subscript permutations. To see how this works, consider the derivation of $aabcbc$:

$$\begin{aligned} V_{abc} &\Rightarrow V_{abc}V_{abc} \\ &\Rightarrow aV_{bc}V_{abc} \\ &\Rightarrow aV_{abc}V_{bc} \\ &\Rightarrow aaV_{bc}V_{bc} \\ &\overset{*}{\Rightarrow} aabcbc. \end{aligned}$$

**3:** An easy argument. Add to the sets of productions $S \to S_1|S_2$, where $S_1$ and $S_2$ are the two start symbols.

**5:** A simple counting argument

$$m(|w|) = |V \cup T|^{|w|}.$$

### Section 11.4

A somewhat general set of exercises that is useful in making the students review the relation between the various language families.

## Chapter 12

The problems in this chapter have characteristics similar to those in Chapter 11. They often have short answers, but they are conceptually difficult and students often have trouble making coherent arguments. It is easy (even for an instructor) to give reasoning that is incorrect or incomplete.

## Section 12.1

**1:** A very straightforward problem. We can add, for instance

$$\delta(q_y, a) = (q_a, a, R)$$
$$\delta(q_a, a) = (q_b, a, R)$$
$$\delta(q_b, a) = (q_a, a, L)$$

for all $a \in \Gamma$.

**2:** A deeper examination of a point treated casually in Theorem 12.1. One need not go into great detail to see that we can always modify the Turing machine so that it saves $w$ first. Then, when it halts, it erases everything except $w$. The argument in Theorem 12.1 is unaffected by any of this, or by any other possible changes in Definition 12.1.

Exercises 3 to 10 rework the basic halting problem, in a way similar to what was done in Examples 12.1 and 12.2. Some students will at first have difficulty, but the more of these they do, the easier it gets.

**4:** This is an instructive exercise that points out that, even if we restrict the requirements, the halting problem may still be undecidable.

Suppose we have an algorithm $A$ that for any given $\widehat{M}$, but a fixed $\widehat{w}$, can decide if $(\widehat{M}, w)$ halts. Then take any $(M, w)$ and modify $M$ so that whenever it halts it checks its input and goes into an infinite loop if this input is not $\widehat{w}$. This is our $\widehat{M}$; we see that $(\widehat{M}, w)$ halts if and only if $(M, w)$ halts and if $w = \widehat{w}$. Now give $(\widehat{M}, \widehat{w})$ to $A$. If $A$ says that this halts, so does $(M, w)$ and vice versa, We have constructed a solution to the original halting problem.

**5:** Given $(M, w)$ we can modify $M$ to $\widehat{M}$ so that if $(M, w)$ halts, $\widehat{M}$ accepts any input. So

$\widehat{M}$ accepts all input implies $(M, w)$ halts

$\widehat{M}$ does not accept all input implies $(M, w)$ does not halt

Again we have a solution to the halting problem.

**6:** Write a special symbol in the starting cell, then use the results of Exercise 3.

**8:** The conclusion is unaffected since as $M_2$ we can always take a finite automaton that accepts $\{a\}$.

**9:** Yes, this is decidable. The only way a dpda can go into an infinite loop is via $\lambda$-transitions. We can find out if there is a sequence of $\lambda$-transitions which (a) eventually produce the same state and stack top and (b) do not decrease the length of the stack. If there is no such sequence, then the dpda must halt. If there is such a sequence, determine whether the configuration that starts it is reachable. Since, for any given $w$, we need only analyze a finite number of moves, this can always be done.

**11:** Although the function is not computable we can still find values for some arguments. It is easy to show that $f(1) = 0$. Since $q_0$ is the only state we cannot have $\delta(q_0, \square)$ defined; otherwise the machine never halts. To compute $f(2)$, note that $\delta(q_0, \square)$ must be defined so that a move to state $q_1$ can be made. But $q_1$ need not be a final state as the machine can halt via undefined transitions. We can then enumerate all possible moves that can be made without repeating a configuration, before this undefined transition is encountered. For example, we can have $\delta(q_0, \square)$, $\delta(q_1, \square)$, $\delta(q_0, 1)$, $\delta(q_1, 0)$ and $\delta(q_0, 0)$ defined, and $\delta(q_1, 1)$ undefined. Then we can make no more than five moves before repeating a configuration. Other options lead to similar results, and we see that $f(2) = 5$.

**12:** Given $(M, w)$, modify $M$ to $\widehat{M}$ so that $\widehat{M}$ accepts its input only if it is $w$. Therefore

$(M, w)$ halts implies that $\widehat{M}$ accepts some input (namely $w$)

$(M, w)$ does not halt implies that $\widehat{M}$ accepts nothing

and we have a solution to the halting problem.

**14:** A simple counting argument based on all the different values that can be taken by $i, j, a, b, D$ in $\delta(q_i, a) = (q_j, b, D)$. Note that $\delta(q_i, a)$ may also be undefined. From this we get

$$m(n) = n|\Gamma|(2n|\Gamma| + 1) = 18n^2 + 3n.$$

**15:** Very similar to Example 12.3 and should not be hard for those who grasp the idea involved there. The maximum number of moves that can be made without repeating a configuration is a bounded function of $b(n)$.

## Section 12.2

**1:** The construction, like many similar ones omitted in the text, is not hard to see, but it takes some effort to spell it all out in detail.

**2:** These are two special instances of Rice's theorem. This should not be too hard for anyone who understands the construction of Theorem 12.4. For example, in (a), modify the original machine so that, whenever it halts, it accepts some string of length five.

**4:** A trap for the unwary: an algorithm is trivial since $L(G)^R$ is always recursively enumerable. Inevitably, there are a number of students who will produce a very complicated "proof" that this is undecidable.

**5:** Contrast this with Exercise 4. This is undecidable in consequence of Rice's theorem.

**6:** A solved problem with a simple answer.

**7:** Again, the same type of argument as in Theorem 12.4 and Example 12.4 (i.e., a version of Rice's theorem), but there are several steps involved that make the problem a little more difficult.

(a) Start with the halting problem $(M, w)$.

(b) Given any $G_2$ with $L(G_2) \neq \varnothing$, generate some $v \in L(G_2)$. This can always be done since $G_2$ is regular.

(c) As in Theorem 12.4, modify $M$ to $\widehat{M}$ so that if $(M, w)$ halts, then $\widehat{M}$ accepts $v$.

(d) From $\widehat{M}$, generate $G_1$ so that $L(\widehat{M}) = L(G_1)$.

Now, if $(M, w)$ halts, then $\widehat{M}$ accepts $v$ and $L(G_1) \cap L(G_2) \neq \varnothing$. If $(M, w)$ does not halt, then $\widehat{M}$ accepts nothing, so that $L(G_1) \cap L(G_2) = \varnothing$. This construction can be done for any $G_2$, so that there cannot exist any $G_2$ for which the problem is decidable.

## Section 12.3

**1:** An easy exercise that helps students understand what is meant by a solution of the Post correspondence problem.

**2:** A straightforward but somewhat lengthy inductive argument.

**4:** In contrast to Exercise 3, this is no longer decidable. This follows from the fact that any alphabet can be encoded with the two-symbol alphabet $\{0, 1\}$.

**5:** These are some minor variations of the Post correspondence problem. In part (a), it follows directly from Theorem 12.1 that this is not decidable. We can have a solution with $w_1, v_1$ on the right if and only if there is one with $w_1, v_1$ on the left. (b) This is also undecidable. If it were decidable,

we could, by trying all possible strings as $w_2$ and $v_2$, get a solution for the modified Post correspondence problem.

**6:** Take the standard Post correspondence problem with $A = (w_1, w_2, ...)$ and $B = (v_1, v_2, ...)$. Construct $A' = (\S, w_1, \S, w_2, ...)$ and $B' = (\S, v_1, \S, v_2, \S, ...)$, where § is some new symbol. Then $(A', B')$ has an even PC-solution if and only if the original pair has a PC-solution.

### Section 12.4

There are many undecidable problems for context-free languages. The exercises in this section give the interested student a chance to explore this issue beyond the very limited treatment given in the text. Unfortunately, most of this is quite difficult and probably should not be given to the average student. For those who wish to tackle more advanced material, this may be suitable. The arguments, many of them lengthy and difficult, can be found in the literature to which we refer.

**1:** One of the more manageable exercises in this section. Note that each $a_i$ occurs only in two productions: $S_A \to w_i S_A a_i$ and $S_A \to w_i a_i$ and that $G_A$ is linear. Thus, as long as we do not terminate the derivation, we have only one choice for the production at each step, determined by the right-most appropriate symbol of the sentential form.

**2 to 5:** Arguments using the Post correspondence problem can be found in Salomaa 1973, p. 281.

**6 to 9:** See Hopcroft and Ullman 1979, p. 201.

## Chapter 13

### Section 13.1

**1:** A routine drill exercise.

Exercises 2 to 7 are instructive as they show the (laborious) way in which complicated functions can be built up from the simple primitive recursive ones. But until students have some experience with this, the exercises are likely to be difficult.

**3:** We can use the primitive recursive functions defined in Examples 13.2 and 13.3 to get

$$equals\,(x, y) = mult\,(subtr\,(1, subtr\,(x, y)), subtr\,(1, subtr\,(y, x))).$$

**4:** Using the results of Exercise 3, we can write

$$f(x,y) = mult(x, subtr(1, equals(x,y))).$$

**5:** The hardest problem in the set. If we define a function *nequals* as the complement of *equals* in Exercise 2, we can write recipes for *rem* and *div*, using a Pascal-like syntax as

$$\begin{aligned} rem(0,y) &= 0 \\ rem(x+1,y) &= (1 + rem(x,y)) * nequals(rem(x,y)+1, y) \\ div(0,y) &= 0 \\ div(x+1,y) &= div(x,y) + equals(rem(x,y)+1, 0). \end{aligned}$$

It is not too hard to rewrite this to be consistent with the notation and requirements of primitive recursive functions.

**6:** The section's one easy problem for primitive recursive functions.

$$\begin{aligned} f(0) &= 1 \\ f(n+1) &= mult(2, f(n)). \end{aligned}$$

Exercises 8 to 14 deal with the classical Ackerman function. An experimental look at Ackerman's function has some interest to students; the rapid growth one can get out of such an innocent-looking recursion is impressive.

**8:** An easy and fun exercise but you can waste a lot of computer time with it.

**9:** Fairly straightforward, but involves induction.

**10 and 11:** From the definitions of $A$ and part (c) of Exercise 9, we get $A(4,0) = 16-3$, $A(4,1) = 2^{16}-3$ and $A(4,2) = 2^{2^{16}}-3$. The general result can then easily be established by induction. For details, see Denning, Dennis, and Qualitz 1978.

**12:** Routine, but tedious.

**13:** To show that Ackerman's function is total on $I \times I$ we must show that the recursion leads to the escape $A(0,y) = y+1$, for all $x$ and $y$. Examining the definition, we see that in each recursive call either $x$ or $y$ is reduced by one. The recursion can therefore not loop indefinitely, but must terminate.

**14:** It is instructive to see how quickly the program comes to a halt with something like a "stack overflow" message.

**15:** Not hard, but useful for getting a better grasp of the $\mu$ operation.

**16:** To bring this in line with the definitions of a primitive recursive function, we can write something like

$$pred(x, y+1) = p_1(y, pred(x, y)).$$

## Section 13.2

**1:** A set of reasonably easy exercises. Some of them illustrate that it is often more convenient to work with Post systems than with unrestricted grammars.

(a)

$$\begin{aligned} A &= \{b, ac\} \\ V_1 b &\to aV_1 b | b \\ aV_2 c &\to abV_2 c | ac \end{aligned}$$

(c)

$$\begin{aligned} A &= \{\lambda, abc, aabbcc\} \\ V_1 abV_2 bcV_3 &\to V_1 aabV_2 bbccV_3. \end{aligned}$$

**2:** Fairly straightforward:

$$V_1 xV_2 \to V_1 axaV_2 | V_1 bxbV_2.$$

**4:** By a simple analogy with Exercise 3

$$L = \left\{ (ab)^{2^n} : n \geq 1 \right\}.$$

**6:** An easy fill-in-the-details exercise.

**7:** It is not hard to generate a few elements of the language and to see the pattern, but a description in set notation seems impossible.

**8:** We can always introduce dummy variables to equalize the number, with the intent of identifying these dummy variables with the empty string. One must be careful, however, that we arrange matters so that these dummy variables cannot be identified with anything else. For example, if in Exercise 2 we just replace

$$V_1 \to V_1 V_1$$

with

$$V_1V_2 \to V_1V_1$$

we change the language. Try instead

$$V_1xV_2 \to V_1V_1x$$

with $x$ some nonterminal constant that is eventually replaced by $\lambda$.

## Section 13.3

**2:** Easy answer: $L = \{a^n b^{2n} a^n\}$.

**3:** $\varnothing$, since $S_2$ can never be reduced to a terminal.

**4:** $abab \Rightarrow Sab \Rightarrow SS \Rightarrow \lambda S$, and the derivation terminates.

**6:** A very difficult problem. We sketch a solution that employs a kind of messenger system often encountered with context-sensitive and unrestricted grammars.

In the first step, we annihilate a $ba$ pair, replacing it with a symbol $x$ that will later become a $b$ again. This is done with the production $ba \to x$. The $x$ is then moved to the left end of the $b$ string by $bx \to xb$. In order to do this before the next $ab$ pair is changed, the second production must be put before the first. The two rules will generate the derivation

$$a^n b^m a^{nm} \overset{*}{\Rightarrow} a^n x^m a^{(n-1)m}.$$

Next, we remove an $a$ and rewrite $x$ as $b$ by

$$ax \to ux$$
$$ux \to bu$$
$$u \to \lambda$$

resulting in

$$a^n b^m a^{nm} \overset{*}{\Rightarrow} a^n x^m a^{(n-1)m} \Rightarrow a^{n-1} b^m a^{(n-1)m},$$

and starting another round by eliminating a $ba$ substring. We still need to provide productions to remove the $b$'s at the end; this is done with $b \to \lambda$. If the input is acceptable, the derivation will end with the empty string; if not, something will be left.

**7:** A simple answer: $a \to a, a \to aa$.

**8:** Although not so easy to see, this is another solution to Exercise 7.

# Chapter 14

## Section 14.1

These questions belong more properly in a course on algorithms and serve as a bridge here.

## Section 14.2

**1:** Count the length of the string, then divide by 2. Write the second part of the input on the second tape, then compare. All of this can be done in $O(n)$ time. For a one-tape machine, the best one can expect is $O(n^2)$.

**2:** In an off-line machine, the input is on a tape. This tape can be used as if it were an input file on a separate track, so none of the algorithms are affected significantly.

**3:** For each move of one, the other makes one move also and there is no extra time in searching for the right place. The only extra effort for a semi-infinite tape is the bookkeeping when we switch from one track to the other, but this does not affect the order of magnitude argument.

**4:** First note that the tape region that can be occupied after $O(n)$ moves has length $O(n)$. The contents of the two tapes can be written on a single tape, one to the left, the other to the right of some marked place. This makes the one-tape machine effectively a machine with semi-infinite tape, but this does not matter. Each move of the two-tape machine can then be simulated making no more than $O(n)$ moves of the read-write head so that the result follows.

**5:** $(x_1 \vee x_3) \wedge (x_2 \vee x_3)$.

**6:** Yes. $x_1 = 1$, $x_2 = 1$, $x_3 = 0$ is one solution.

**7:** The answer is yes in both cases. This shows that the second algorithm is more efficient.

## Section 14.3

**1:** Straightforward, fill-in-the-details exercise.

**2 and 3:** Here we continue the idea suggested in Exercise 2, Section 14.2. To divide input into three equal parts, scan it and produce a marker for each third symbol. The number of markers then determines the value of $|w|/3$.

**4:** Follows directly from Theorem 14.2.

## Section 14.4

**1:** An easy argument.

**2:** Useful if you want students to go a little deeper into **NP**-completeness.

**3:** We don't know the answer, but it is a problem with a single instance, so it must be decidable.