Introducing the Theory of Computation

Wayne Goddard

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
6339 Ormindale Way
Mississauga, Ontario L5V 1J2
Canada

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

Jones and Bartlett's books and products are available through most bookstores and online booksellers. To contact Jones and Bartlett Publishers directly, call 800-832-0034, fax 978-443-8000, or visit our website www.jbpub.com.

Substantial discounts on bulk quantities of Jones and Bartlett's publications are available to corporations, professional associations, and other qualified organizations. For details and specific discount information, contact the special sales department at Jones and Bartlett via the above contact information or send an email to specialsales@jbpub.com.

Production Credits
Acquisitions Editor: Tim Anderson
Production Director: Amy Rose
Editorial Assistant: Melissa Elmore
Senior Marketing Manager: Andrea DeFronzo
Manufacturing Buyer: Therese Connell
Composition: Northeast Compositors
Cover Design: Kate Ternullo
Cover Image: © Cindy Hughes/ShutterStock, Inc. and © Andreas Nilsson/ShutterStock, Inc.
Printing and Binding: Malloy, Inc.
Cover Printing: Malloy, Inc.

Library of Congress Cataloging-in-Publication Data
Goddard, Wayne.
Introducing the theory of computation / Wayne Goddard. — 1st ed.
p. cm.
Includes bibliographical references and index.
ISBN-13: 978-0-7637-4125-9
ISBN-10: 0-7637-4125-6
1. Machine theory. 2. Computational complexity. I. Title.
QA267.G57 2008
511.3'5—dc22
2007049462

6048
Printed in the United States of America
12 11 10 09 08 10 9 8 7 6 5 4 3 2 1

To Stuart, friend and collaborator forever

Preface

Instructions for using this book: Read, think, repeat.

To the Instructor

This is a text for an undergraduate course in the theory of computation; it is also appropriate for courses in automata theory and formal languages. The text covers the standard three models of finite automata, grammars, and Turing machines, as well as undecidability. An introduction to time and space complexity theories and a separate part on the basics of complexity theory are also included.

Goals and Features

When I wrote this text, I focused on the standard material for a course in the theory of computation or automata theory. As a result, this text provides a concise introduction to core topics taught in a course on either of these subjects.

One difference between this text and others on the subject is the use of flowcharts for the pushdown automata. Another difference is that although the material is undoubtedly mathematical, I have tried to reduce the use of mathematical notation. Additionally, this text incorporates the following:

- an engaging, student-friendly writing style that moves through material at a pace appropriate for undergraduate students
- a wide range of problems, varying in level of difficulty, which allows students to test themselves on key material covered in the given chapter

Solutions to selected exercises are in the appendix and are noted with the symbol ★. The most difficult exercises are labeled with the symbol Ⓗ.

Full solutions are provided in the Online Instructor's Manual, which is available online through the Jones and Bartlett website at: http://www.jbpub.com/catalog/9780763741259/.

Organization

The text is divided into five parts: Regular Languages, Context-Free Languages, Turing Machines, Undecidability, and Complexity Theory. The final chapter in each of the five parts can be viewed as optional—specifically, Chapters 5, 10, 16, and 19. These chapters provide additional information but can be omitted without any impact on the overall course. I include more material than necessary for a single semester course, which provides instructors with the freedom to structure their course and omit or include whichever relevant topics they choose.

To the Student

Welcome to the theory of computation! The material is theoretical, although the early stages are less so. Part IV, Undecidability, is both theoretical and challenging.

As you work through the text, do not lose yourself in the theoretical details. Remember the bigger picture! The finite automata and grammars we see in the first few parts are two of the most efficient and successful techniques in computer programming. The Turing machines and later material show that there are limits to what computer programming can do, even if the actual boundaries are not yet clear. There are problems, procedures, programs, and paradoxes. I encourage you to read and re-read the more difficult sections for better understanding.

The exercises are at varied levels of difficulty. Exercises that are more challenging are marked with the symbol Ⓗ. Solutions to problems marked with the symbol ★ are provided in the appendix.

Acknowledgments

I was very fortunate to have many people help me write and publish this text. Thank you to my professor at MIT, Michael Sipser, for his instruction. Thanks to family and friends, past and present. In particular, thanks go to Steve Hedetniemi for offering some of his problems and to my students at Clemson University who class-tested early drafts of the text.

Thanks also to the readers and reviewers whose comments greatly improved the book:

Petros Drineas, Rensselaer Polytechnic Institute
Stephen T. Hedetniemi, Clemson University
K. N. King, Georgia State University
Anne-Louise Radimsky, California State University, Sacramento
Neil W. Rickert, Northern Illinois University
R. Duane Skaggs, Morehead State University
Nancy Lynn Tinkham, Rowan University
Jinhui Xu, State University of New York at Buffalo

I want to express my gratitude to the staff at Jones and Bartlett Publishers for their hard work on this text. Thank you to Tim Anderson, Acquisitions Editor; Amy Rose, Production Director; and Melissa Elmore, Editorial Assistant.

Wayne Goddard
Clemson SC

Contents

part

1

Regular Languages

This book is about the fundamental capabilities and ultimate limitations of computation. What can be done with what abilities.

We will see three main models of a computer: a finite automaton, a pushdown automaton, and a Turing machine. In parallel with that we will see other formal ways to describe computation and algorithms, through the language of mathematics, including regular expressions and grammars. In the last two parts of the book, we take our computer and ask what can be solved, and if it can be solved, what resources are required, such as speed and memory. Concepts such as finite automata are certainly useful throughout computer science, but even proving something impossible is good because it tells you where not to look, that you have to compromise on some aspect.

The input to our computers is always strings. We discuss this later, but it is true that everything can be converted to questions about strings.

We start with the simplest form of computer, or maybe, machine. For example, an automatic door. It spends all day either open or closed. The design is simple. Open, closed. Or maybe opening, open, closing, closed. Or maybe there's an override. This is the simplest form of a machine: only internal memory, nothing external, just reacting to events. Ladies and gentlemen, I give you thé finite automaton.

Any language is necessarily a finite system applied with different degrees of creativity to an infinite variety of situations, . . . —David Lodge

Nature is a self-made machine, more perfectly automated than any automated machine. —Eric Hoffer

Mathematics, rightly viewed, possesses not only truth, but supreme beauty—a beauty cold and austere, like that of sculpture, without appeal to any part of our weaker nature, without the gorgeous trappings of painting or music, yet sublimely pure, and capable of a stern perfection such as only the greatest art can show. —Bertrand Russell

I hate definitions. —Benjamin Disraeli

chapter

1

Finite Automata

The most basic model of a computer is the finite automaton. This is a computer without memory; or rather, the amount of memory is fixed, regardless of the size of the input.

1.1 A Finite Automaton Has States

A **string** is a sequence of characters or **symbols**. A finite-state machine or **finite automaton** (FA) is a device that recognizes a collection of strings. (The plural of automaton is automata.) An FA has three components:

1. An **input tape**, which contains a single string
2. A sensor or **head**, which reads the input string one symbol at a time
3. **Memory**, which can be in any one of a finite number of states—so we speak of the **current state** of the automaton

The "program" of the FA prescribes how the symbols that are read affect the current state. The **final state** for a string is the state the automaton is in when it finishes reading the input.

Operating an FA

1. Set the machine to the start state.
2. If end-of-string then halt.
3. Read a symbol.
4. Update the state according to current state and symbol read.
5. Goto step 2.

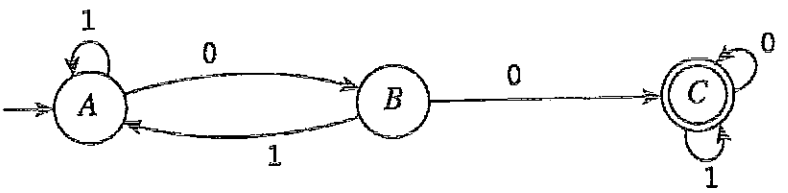
An FA can be described by a diagram. In the diagram, each state is drawn as a circle; we sometimes name a state by putting its name inside the

circle. Each state has, for each symbol, an arrow showing the next state. The initial or **start state** is shown by an arrow into it from no state.

The purpose of an FA is as a **recognizer**—essentially, it acts like a boolean function. For any FA, certain states are designated as **accept** states and the remainder are **reject** states. An **accept state** is indicated by a double circle in the diagram.

Definition An FA accepts the input string if the final state is an accept state, otherwise it rejects the input string.

Example 1.41 The following is an FA with 3 states called A, B, and C. The start state is A, and C is the only accept state.



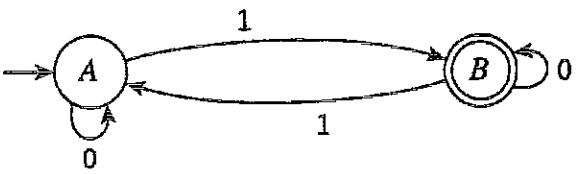
Consider its behavior when the input string is 101001:

Current State	Symbol Read	New State
A	1	A
A	0	B
B	1	A
A	0	B
B	0	C
C	1	C

Here, the final state is C. Similarly, the final state for 11101 is A, and for 0001 it is C.

What does it take to get to the accept state? This machine accepts all strings of 0's and 1's with two consecutive zeroes somewhere.

Example 1.2 Consider the following FA.



This FA accepts strings like 100 and 0101001 and 11111, and rejects strings like 000 and 0110.

Can you describe exactly which strings this FA accepts?

If you can't wait, then read on. This FA ignores the symbol 0 (it doesn't change state). It only worries about the symbol 1; here it alternates states. The first 1 takes it to state B, the second 1 takes it to state A, the third to state B, and so on. So it is in state B whenever an odd number of 1's have been read.

That is, this machine accepts all strings of 0's and 1's with an odd number of 1's.

In these examples, we simply use letters for the names of states. Sometimes you can find more descriptive names.

1.2 Building FAs

There is no magic method for building FAs. It takes practice and thinking (though some of the machinery in subsequent chapters will be helpful). In this section, we consider how to build an FA for a specific purpose.

First, we need a few more definitions. An **alphabet** is a set of symbols. A **language** is a set of strings, where the strings have symbols from a specific alphabet. The language of an FA is the set of strings it accepts. For example, the language of the first FA from Example 1.1 is the set of all strings with alphabet $\{0, 1\}$ that contain the substring 00.

We often use Σ to denote the alphabet. Often the alphabet will be $\{a, b\}$ or $\{0, 1\}$; though this is abusing the term, we refer to strings from the alphabet $\{0, 1\}$ as **binary** strings. A **unary language** is one where the alphabet has only one symbol.

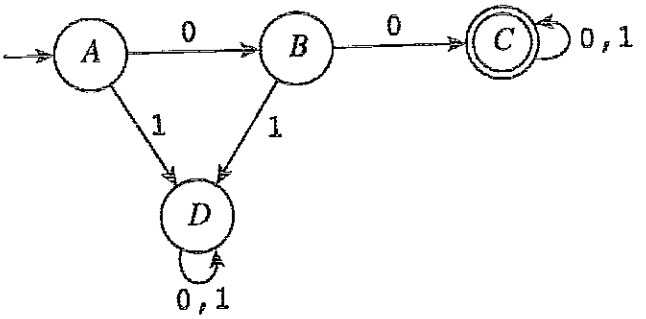
The **length** of a string is the number of symbols in it. The **empty string** has length 0: it is a string without any symbols and is denoted ε.

Sometimes the obvious natural idea works.

Example 1.3

All binary strings starting with 00.

The approach is to read the first two symbols, and that gives one the answer. The rest of the string is ignored (which corresponds to the automaton never changing state).



To save clutter on the diagram, you can label an arrow with multiple symbols separated by commas.

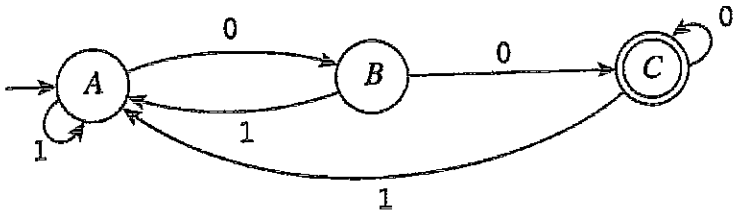
If you are interested in the final two symbols, then you have to work a little harder, as we see next.

Example 1.4

All binary strings ending in 00.

Well, the first thing you might do is to think about what happens if the first two symbols are 0. That must take you to an accept state. So, there are at least three states: A, B, and C, where A is the start state, C is an accept state, and a 0 takes you from A to B, and from B to C.

But what happens if you read a 1? Well, if you think about it, this should take the FA back to the start state, no matter what state it is in. Another question is, what happens if you get a 0 in state C? Well, staying in C seems reasonable. This yields the following FA:



Does this really work? Yes. Perhaps it helps to think about what exactly it means to be in each state. State C means that the previous two symbols were 00, state A means that the previous symbol was 1, and state B means that the last two symbols were 10.

A common type of state is a **trap**. This is a state that, once entered, you can never leave. For example, state C of Example 1.1 is a trap. How can you recognize a trap from the picture? *A trap has no arrow out.* Traps can be used in two ways:

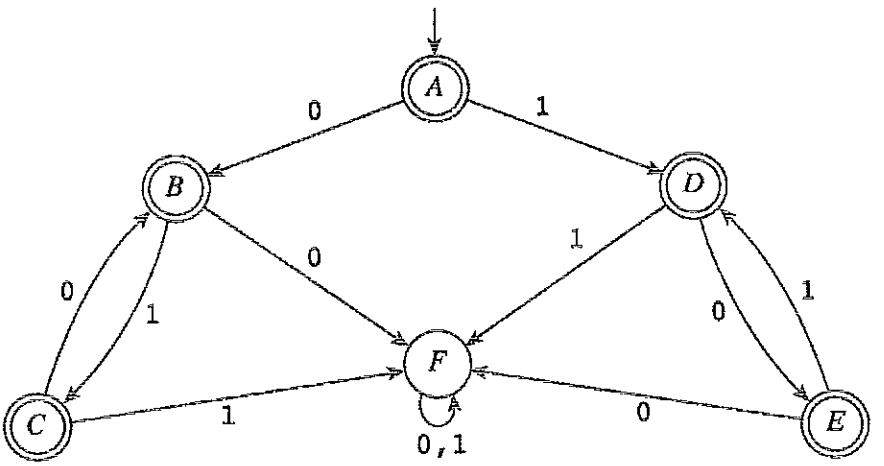
a) As a reject state for partly read strings that will never be accepted
b) As an accept state for partly read strings that will definitely be accepted

Example 1.5

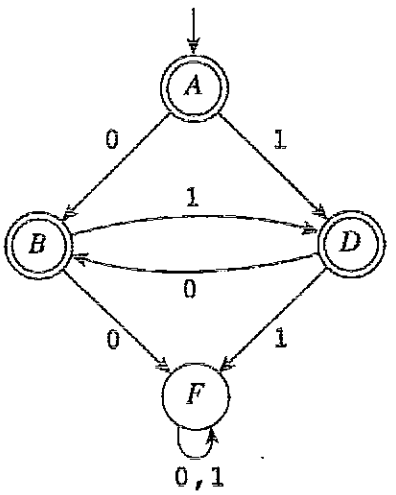
An FA that accepts all binary strings where 0's and 1's alternate.

The main idea is a pair of states that oscillate: 0 takes you to one and 1 takes you back again. We also need a trap if ever two consecutive symbols are the same.

One approach is to have a start state that splits into two sub-machines. This gives the following solution:



But actually, you can do it with fewer states:



So, although a given FA corresponds to only one language, a given language can have many FAs that accept it.

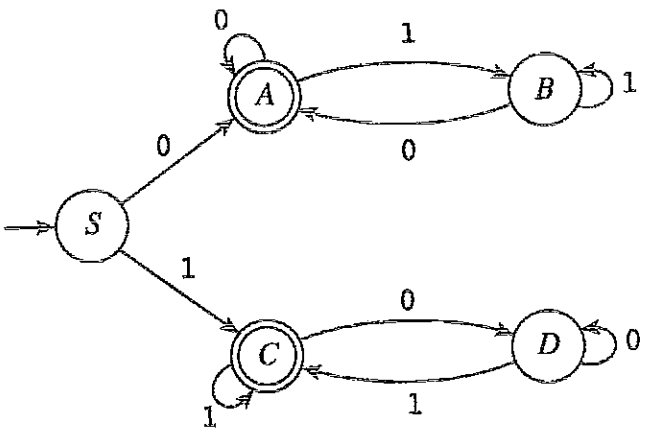
Note that you must always be careful about the *empty string*: should the FA accept ε or not. (Though, to be fair, this is not always clear from the English description.) In the preceding example, the empty string is accepted because the start state is also an accept state.

Another useful technique is **remembering specific symbols**. In the next example, you must forever remember the first symbol; so the FA splits into two pieces after the first symbol.

Example 1.6

An FA that accepts all binary strings that begin and end with the same symbol.

It is clear that the automaton must be in two different states based on the first symbol that is read. After that, you need to keep track of the current symbol just read, in case it turns out to be the final symbol. But that is enough.



Note that there must not be any connection between the top and bottom halves, since otherwise the FA might forget the first symbol read.

And now:

For You to Do!

Give an FA for each of the following three languages:

1. All binary strings with at least one 0
2. All binary strings with at most one 0
3. All binary strings starting and ending with 0 (and the string with a single 0 counts)

The solutions to the practice problems are given at the end of the chapter.

1.3 Representing FAs

A diagram is not the only way to represent an FA. Indeed, to feed an FA into a computer you must have another way to represent it. One way to go is called a **transition table**: this is a matrix that lists the new state given the current state and the symbol read.

Example 1.6 (continued)

Here is the transition table for the FA that accepts all binary strings that begin and end with the same symbol.

		Input	
		0	1
State	S	A	C
	A	A	B
	B	A	B
	C	D	C
	D	D	C

We shall see soon a more general type of FA; the FA we have seen so far is actually a deterministic one (sometimes abbreviated DFA). One can provide a **formal definition** of a DFA using the language of mathematics. We may define a deterministic finite automaton as a 5-tuple $(Q, \Sigma, q_0, T, \delta)$ where:

- Q is a finite set of states
- Σ is an alphabet of input symbols
- q_0 is the start state
- T is a subset of Q giving the accept states
- δ is the **transition function** that maps a state and symbol to a state. (In mathematical notation, we say that $\delta\colon Q \times \Sigma \mapsto Q$.) For example, $\delta(r, 1) = s$ means that in state r on reading symbol 1 change to state s.

This approach makes precise what we need to specify for an FA. It also allows us to use mathematical notation to describe one.

Example 1.6 (continued)

And for the previous example, the 5-tuple is $(Q, \Sigma, q_0, T, \delta)$ where

$$Q = \{S, A, B, C, D\}$$
$$\Sigma = \{0, 1\}$$

$q_0 = S$
$T = \{A, C\}$
δ is given by the previous transition table

EXERCISES

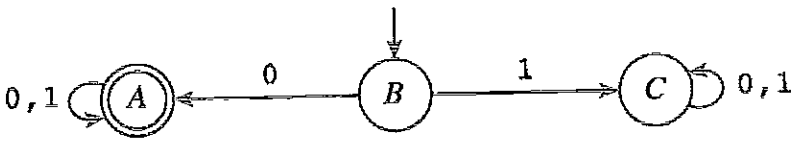
1.1 Give an FA that accepts only the string 0110.

1.2 Give an FA that accepts only binary strings of length 3.

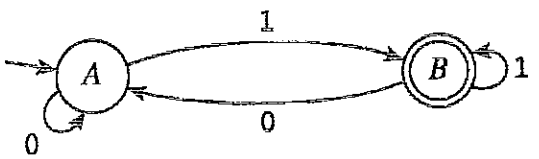
1.3 If in Example 1.1 we make both states B and C accept states, describe in English the strings the FA accepts.

★ **1.4** For the following two FAs, determine which of the strings 0110, 1, 1011010, and 00000 are accepted.

a)



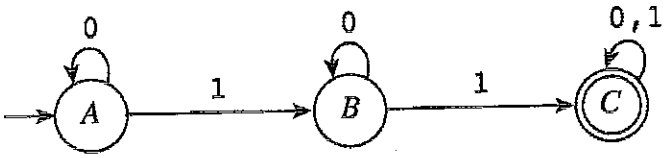
b)



1.5 For the preceding two FAs, describe in English the strings each accepts.

1.6 Draw an FA that accepts any string of 0's and 1's.

1.7 For the following FA, determine which of the strings 0110, 1, 1011010, and 00000 are accepted.



★ **1.8** For the preceding FA, describe in English the strings it accepts.

1.9 Give an FA for each of the following languages:

a) All binary strings with at least three 1's
b) All binary strings with an odd number of 1's
c) All binary strings without 111 as a substring
d) All binary strings where every *odd* position is a 1

1.10 We can use a spreadsheet program to simulate an FA. This involves storing the FA's transition table as a lookup table. Describe how to simulate an FA using a spreadsheet program.

1.11 Construct an FA that accepts all binary strings with precisely three 1's.

★ **1.12** Construct an FA that accepts all strings of {a, b, c} that contain an odd number of a's.

1.13 Give an FA for the language of all binary strings that have at least three symbols and whose first and last symbols are different.

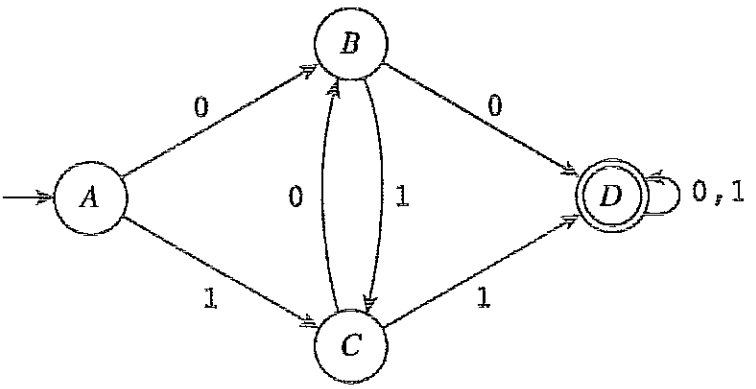
1.14 Give an FA whose language is the set of strings of a's, b's, and c's that contain abc as a substring.

1.15 Construct an FA that accepts all strings of {a, b} that contain either ab or bba (or both) as substrings.

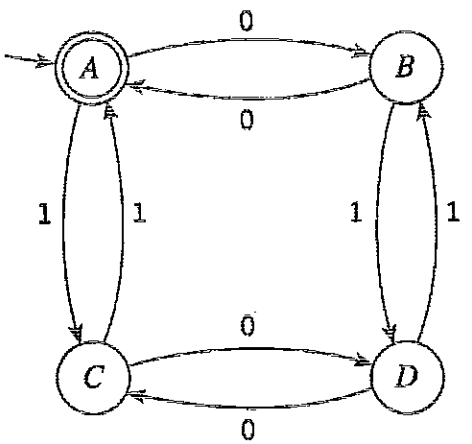
★ **1.16** Construct an FA that accepts all strings of {a, b, c} whose symbols are in alphabetical order. (For example, aaabcc and ac are okay; abca and cb are not.)

Ⓗ **1.17** Construct an FA that accepts all binary strings with an even number of 0's and the number of 1's is a multiple of 3.

1.18 Explain in English what the following FA accepts:

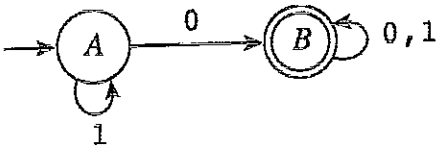


1.19 Explain in English what the following FA accepts:

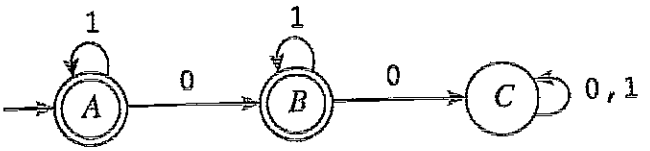


"For You to Do" Exercise Solutions

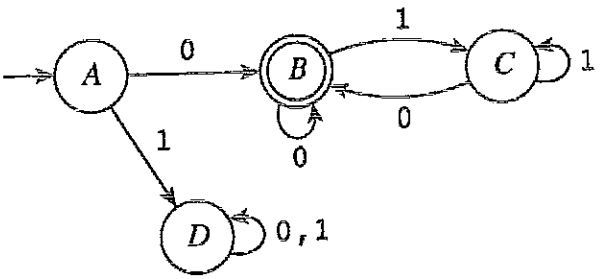
1.



2.



3.



Chapter 2

Regular Expressions

Often you are faced with the task of recognizing strings of a certain type. For example, suppose you want a method or procedure that recognizes decimal numbers. To write such a method, we need to be clear about what we mean by a decimal number. We can define a decimal number as follows: "Some digits followed maybe by a point and some more digits."

That definition is far too sloppy to be of use. The following is better: "optional minus sign, any sequence of digits, followed by optional point, and if so then optional sequence of digits." To be more precise, we can write the description of a decimal number as a regular expression:

$$(- + \varepsilon)\,\mathtt{D}\,\mathtt{D}^*\,(\varepsilon + .\,\mathtt{D}^*)$$

where D stands for a digit. But then you need to understand what a regular expression is . . . .

2.1 Regular Expressions

A **regular expression** (RE) corresponds to a set of strings; that is, a regular expression describes a language. It is built up using the three regular operations called **union**, **concatenation**, and **star**, described in the following paragraphs. A regular expression uses normal symbols as well as four special symbols:

$$+ \quad ^* \quad (\quad)$$

A regular expression is interpreted using the following rules:

- The parentheses (and) are used for grouping, just as in normal math.
- The plus sign (+) means **union**. Thus, writing

  $0+1$

  means either a zero or a one. We also refer to the + as **or**.
- The **concatenation** of two expressions is obtained by simply writing one after the other without spacing between them. For example,

  $(0+1)\,0$

  corresponds to the pair of strings 00 and 10.
- The symbol ε stands for the empty string. Thus, the regular expression

  $(0+1)\,(0+\varepsilon)$

  corresponds to the four strings 00, 0, 10, 1.
- The asterisk (*) is pronounced *star* and means zero or more copies.

  For example, the regular expression

  a^*

  corresponds to any string of a's: $\{\varepsilon, a, aa, aaa, \ldots\}$. Also,

  $(0+1)^*$

  corresponds to all binary strings. (It's $\varepsilon + (0 + 1) + (0 + 1)(0 + 1) + \ldots$)

Example 2.1 The regular expression

$(01)^*$

corresponds to the set containing ε, 01, 0101, 010101, and so forth.

Example 2.2 What about a regular expression for the language of all binary strings of length at least 2 that begin and end in the same symbol?

This is given by:

$0(0+1)^*0 + 1(0+1)^*1$

Note too the **precedence** of the regular operators: star first, then concatenation, then or. That is, the *star* always refers to the smallest piece it can, the *or* to the largest.

Example 2.3

Consider the regular expression

$$((0+1)^*1+\varepsilon)\ (00)^*\ 00$$

The language of this RE is the collection of all binary strings that end with an even nonzero number of 0's.

There are snares waiting for the unwary. For example, someone proposed the RE

$$(0+1)^*\ (00)^*\ 00$$

for this language. But this RE corresponds to any binary string that ends with 00: take any string you like from the $(0+1)^*$ part, take nothing from the $(00)^*$ part, and 00 from the last part.

In general, if you form an RE by the **or** of two REs, call them R and S, then the resulting language is the union of the languages of R and S. This is why $+$ is called the union operation.

If you form an RE by the **concatenation** of two REs, call them R and S, then the resulting language consists of all strings that can be formed by taking one string from the language of R and one string from the language of S and concatenating them.

If you form an RE by taking the star of an RE R, then the resulting language consists of all strings that can be formed by taking any number of strings from the language of R (they need not be the same and they need not be different), and concatenating them.

Indeed, we can talk about the union, concatenation, or star of languages. For example, here is a recursive definition of the star of a language:

Recursive definition of L^*

1. $\varepsilon \in L^*$.
2. If $x \in L^*$ and $y \in L$ then $xy \in L^*$.

Example 2.4

If language L is {ma, pa} and language M is {be, bop}, then

$L+M$ is {ma, pa, be, bop}.
LM is {mabe, mabop, pabe, pabop}.
L^* is {ε, ma, pa, mama, ... , pamamapa, ...}.

We also use the following notation.

Notation	If Σ is some alphabet, then Σ^* is the set of all strings using that alphabet.

2.2 Kleene's Theorem

It is not hard to write a recognizer to accept a string if and only if the string is of the required form described by the preceding example regular expressions. But what about a complicated regular expression? Fortunately, there is an algorithm (and hence a program) for doing this. In fact, you can always build a finite automaton.

Example 2.5 Give an FA for the language of the RE

$$(0+1)^*00(0+1)^*$$

This was given in Example 1.1.

In the next chapter we will prove the following:

Kleene's Theorem	There is an FA for a language if and only if there is an RE for the language.

This theorem is not just of theoretical benefit: for, it is often easy to describe a recognition problem by an RE, and it is easy to write a procedure that simulates an FA. The theorem, translated into an algorithm, gives one the bridge to convert an RE into an FA. We will use the term **regular language** for a language that is accepted by some FA or described by an RE.

2.3 Applications of REs

We mention here a few of the applications of finite automata and regular expressions. Several more are discussed in Chapter 5.

Regular expressions are used in the design of compilers to describe some of the pieces of the language. For example, in Pascal or Java an integer has a certain form, a real number has a certain form, and so forth. The RE for

each piece can be automatically converted into an FA that recognizes them. A scanner or **tokenizer** is a program that scans the input and determines and identifies the next piece. A tokenizer is provided as a standard part of Java and Unix.

Regular expressions can also be used for searching a file. For example, suppose in a long document you've written hexadecimal numbers with an h behind, such as 273h or 22h, and your boss now says that hexadecimal numbers should be written with a # in front, such as #273 or #22. Well, if you have a good search-and-replace option (such as in Perl), you can type something like

```
find: ([0123456789]+)h
replace with: #\1
```

In the find text, the symbol + is like * and stands for one or more copies; the square brackets specify a set of symbols; and the parentheses specify an expression. The \1 in the replacement text means the first expression in the find text.

For You to Do!

Give an RE for each of the following three languages:

1. All binary strings with at least one 0
2. All binary strings with at most one 0
3. All binary strings starting and ending with 0

EXERCISES

2.1 For each RE, state which of the following strings is in the language of the RE: ε, abba, bababb, and baaaa.

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

2.2 For each RE, give two strings that are in the corresponding language and two strings that are not:

a) $a(a+b)^*b$
b) $a^*a+\varepsilon+b^*$
c) $(ab+ba)^*$

2.3 Give REs for:

a) All binary strings with exactly two 1's
b) All binary strings with a double symbol (contains 00 or 11) somewhere
c) All binary strings that contain both 00 and 11 as substrings
d) All binary strings without a double symbol anywhere

★ **2.4** Give an FA for decimal numbers as described at the start of this chapter.

2.5 Give an RE for the language of all binary strings of length at least two that begin and end with the same symbol.

2.6 Give REs and FAs with alphabet $\{a, b\}$ for

a) All strings containing the substring aaa
b) All strings not containing the substring aaa
c) All strings that do not end with aaa
d) All strings with exactly 3 a's.
e) All strings with the number of a's divisible by 3.

2.7 Give an RE for the language of Exercise 1.12.

★ **2.8** Give an RE for the language of Exercise 1.13.

2.9 Give an RE for the language of Exercise 1.14.

2.10 Give an RE for the language of Exercise 1.15.

2.11 Give an RE for the language of Exercise 1.16.

★ **2.12** Give an RE for the language of Exercise 1.19.

2.13 Give a regular expression for the complement of the RE 111 with respect to the alphabet $\{0, 1\}$. That is, the RE should allow every possible binary string except for the string 111.

2.14 Does every regular language correspond to only *one* RE? Does every RE correspond to only *one* regular language? Discuss.

2.15 Show that the language of RE $(0^*1^*)^*$ is all binary strings.

★ **2.16** Simplify the following REs (that is, find a simpler RE for the same language):

a) $(r+\varepsilon)^*$
b) $ss^*+\varepsilon$
c) $(\varepsilon+r)(r+s)^*(\varepsilon+r)$
d) $(r+s+rs+sr)^*$

2.17 In a string, a **block** is a substring in which all symbols are the same and which cannot be enlarged. For example, 0001100 has *three* blocks. Let L be the language consisting of all binary strings such that every block has length 2 or 3. Give both an FA and an RE for L.

Ⓗ **2.18** Let C_n denote the set of all binary numbers that are a multiple of n. Show that C_7 is regular. (Hint: Find an FA.)

3.19 Propose a formal definition of a regular expression in the spirit of the one given for an FA on Page 9.

"For You to Do" Exercise Solutions

1. $(0+1)^*0(0+1)^*$
2. $1^*+1^*01^*$
5. $0(0+1)^*0+0$

In each case several answers are possible.

chapter 3

Nondeterminism

The FAs we defined—and our real-world computers—are deterministic: the program completely specifies a unique action at each stage. But what happens if we allow the machine some flexibility or nondeterminism? And this is not just a theoretical musing: the conversion from RE to FA cannot be performed without introducing nondeterminism.

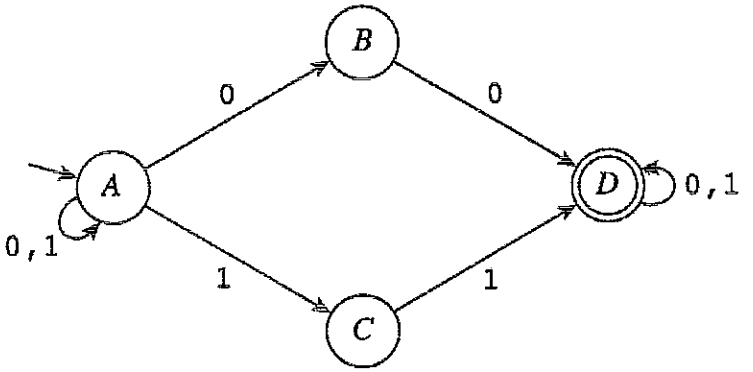
3.1 Nondeterministic Finite Automata

We define a **nondeterministic** finite automaton as follows. In this automaton, for each state there can be zero, one, two, or more transitions corresponding to a particular symbol of the alphabet. If the automaton gets to a state where there is more than one possible transition corresponding to the input symbol, we say that the automaton **branches**. If the automaton gets to a state where there is no such transition, then that branch of the automaton halts and rejects: we say that it **dies**. Thus, it is possible that some branch accepts and some branch rejects. How do you know whether the string is in the language or not? We say the following.

Definition	A nondeterministic FA (NFA) **accepts** the input string if **there exists** some choice of transitions that leads to an accept state.

Note that this definition is *not* symmetric: one accepting branch is enough for the overall automaton to accept, but every branch must reject for the overall automaton to reject. Also, this is only a model of a computer: nondeterministic machines do not actually exist (though some would disagree).

Example 3.4 What does this NFA accept?

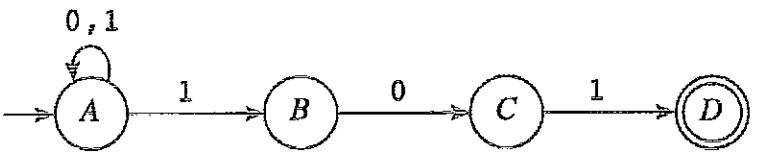


For example, 0110 is accepted because the automaton can proceed $A\,A\,C\,D\,D$. But 101010 is not accepted; any branch that goes to states B or C dies.

In fact, this NFA accepts any binary string that contains 00 or 11 as a substring.

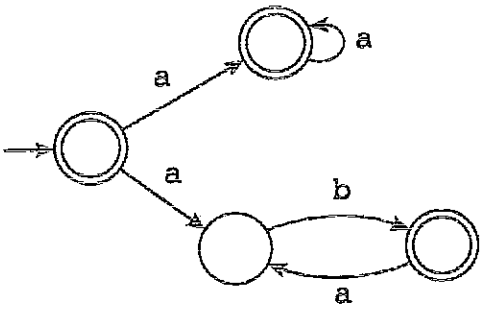
We write **DFA** when we want to specify a deterministic finite automaton, the one defined earlier. If the type doesn't matter, we just write **FA** again.

Example 3.5 An easy nondeterministic automaton to build is one for the language of all strings with a particular ending. Here is an NFA that accepts all binary strings that end with 101. To accept strings in the language, the machine stays in state A until it nondeterministically guesses to move to state B at the correct 1.



In the following example, we use nondeterminism to look for two patterns "simultaneously."

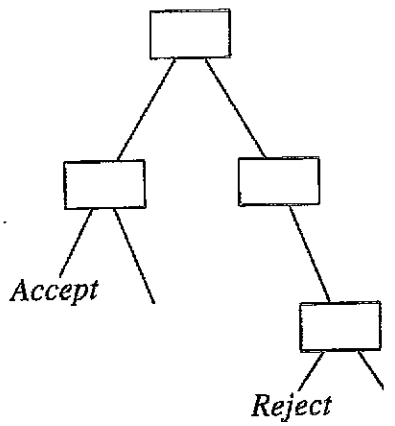
Example 3.3 **An NFA for $a^* + (ab)^*$**



3.2 What Is Nondeterminism?

There are many ways to view nondeterminism. You can think of the machine as guessing the next move: the "**guess and verify**" idea. In this, you assume that the machine is clairvoyant and always guesses correctly the next state to go to. However, the rules are that the machine may make as many guesses as it likes, but it must "check" them. In Example 3.1, the automaton guesses whether the start of the double symbol has been reached or not; it checks its guess by reading the next two symbols. If the string is not in the language, then no sequence of guesses can lead to acceptance.

Alternatively, you can think of nondeterminism as a **computation tree** growing downward. At each node the children are all the possible successors. You may then talk about a branch of the computational tree: the string is accepted exactly when one of the branches ends in an accept state.



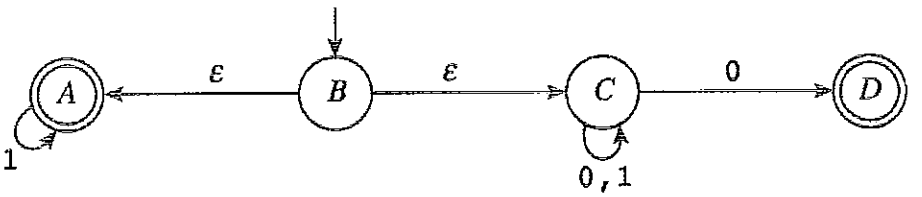
This implies that you can (in some sense) simulate an NFA using guess and backtracking—more on this later.

3.3 ε-Transitions

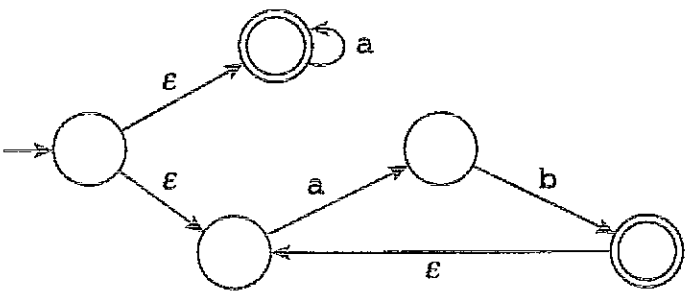
We also allow ε-transitions: arrows labeled with the empty string. These allow the machine to change state without consuming an input symbol. For example, if the overall language is the union of two languages, such a transition allows the machine to guess which part the input string is in.

Example 3.4 What about an NFA that accepts all binary strings where the last symbol is 0 or which contain only 1's?

Here is one:



Example 3.5 Here is another NFA for $a^* + (ab)^*$, which was originally considered in Example 3.3:



We can provide a similar **formal definition** for an NFA. It is a 5-tuple $(Q, \Sigma, q_0, T, \delta)$ where as before:

- Q is a finite set of states.
- Σ is an alphabet of input symbols.
- q_0 is the start state.
- T is a subset of Q giving the accept states.
- δ is the transition function.

The difference is that now the transition function specifies a *set* of states rather than a state: it maps $Q \times (\Sigma \cup \{\varepsilon\})$ to { subsets of Q }. For example,

in the NFA of Example 3.4, you have $\delta(A, 1) = \{A\}$, $\delta(B, \varepsilon) = \{A, C\}$, and $\delta(D, 0) = \emptyset$.

For You to Do!

1. Give an NFA for the set of all binary strings that have either the number of 0's odd, or the number of 1's not a multiple of 3, or both.

3.4 Kleene's Theorem Revisited

Surprisingly perhaps, nondeterminism does not add to the power of a finite automaton.

Kleene's Theorem

The following are equivalent for a language L:

1. There is a DFA for L.
2. There is an NFA for L.
3. There is an RE for L.

This theorem is proved in three steps. In fact, we produce an algorithm that does the conversion:

$$(3) \Longrightarrow (2) \Longrightarrow (1) \Longrightarrow (3).$$

Thus, if you know that a language is regular, then you know that it is accepted by some DFA, by some NFA, and described by some RE.

3.5 Conversion from RE to NFA

We consider here the conversion from RE to NFA. This is a recursive construction.

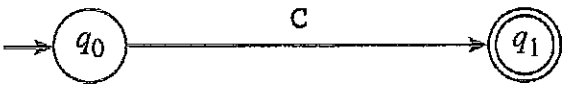
Because an RE is built up recursively, we need to (1) describe an NFA for each symbol and for ε; and (2) show that if there exists NFAs for REs A and B, then there exist NFAs for $A+B$, AB, and A^*. (Why will this constitute a proof?)

Converting from RE to NFA (Outline)

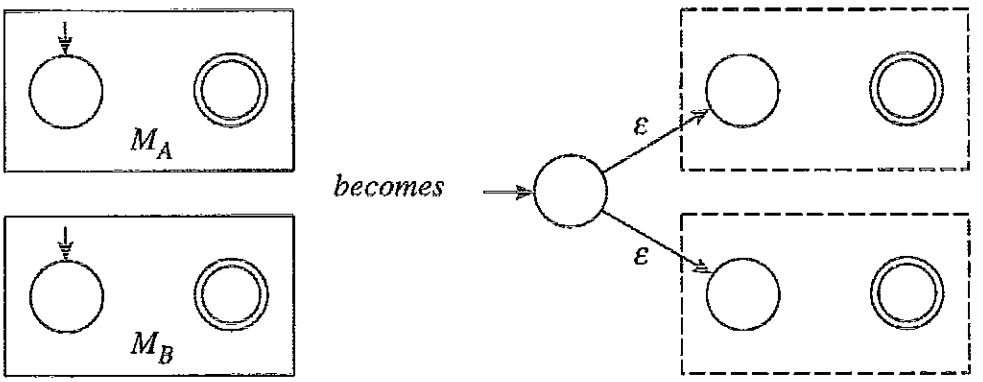
0. If the RE is the empty string, then output simple NFA.
1. Else if the RE is a single symbol, then output simple NFA.
2. Else if RE has form $A+B$, then combine the NFAs for A and B.
3. Else if RE has form AB, then combine the NFAs for A and B.
4. Else if RE has form A^*, then extend the NFA for A.

What remains is to work out how to combine or extend these NFAs.

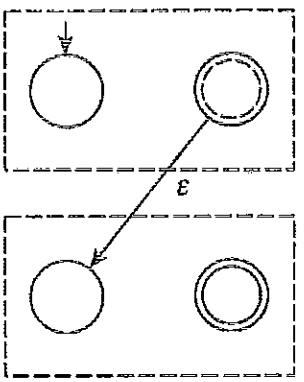
1) An NFA for a single symbol C consists of two states q_0 and q_1. The first is the start state and the second the accept state. There is one transition from q_0 to q_1 labeled with that symbol:



2) Armed with an NFA M_A for A and M_B for B, here is one for $A+B$. Add a new start state with ε-transitions to the original start states of both M_A and M_B. The machine guesses which of A or B the input is in.



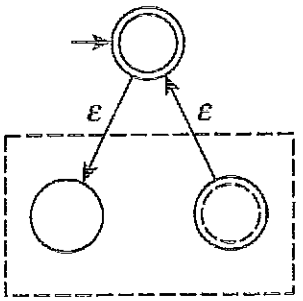
3) Here is one for the concatenation AB. Start with NFAs M_A and M_B. The first step is to put ε-transitions from the accept states of M_A to the start state of M_B. Then, you must make the original accept states of M_A reject.



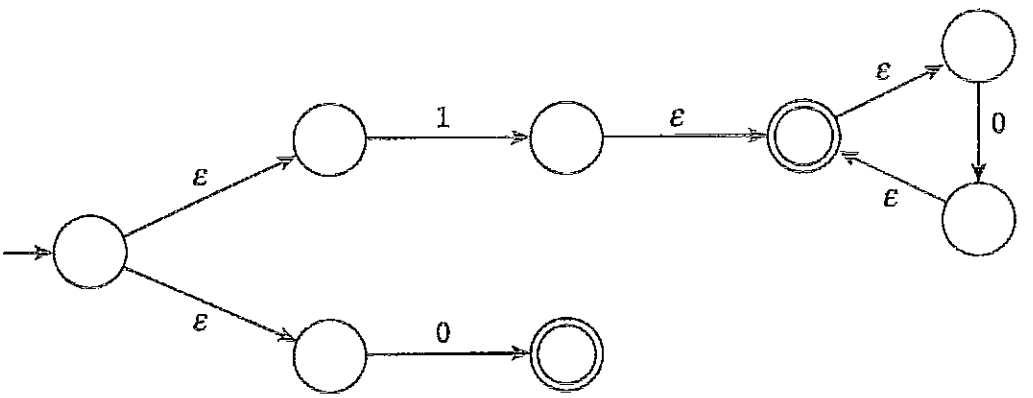
4) Here is one for A^*. The idea is to allow the machine to cycle from the accept state back to the start state. The natural try is to make the start state accepting and put ε-transitions from the accept states back to the start state. But this doesn't quite work. Why? Well, that might add

strings to the language (that previously ended in the start state which wasn't accepting).

What you can do is build a new start state, which is the only accept state; then put an ε-transition from it to the old start state and from the old accept states to it; and change every old accept state to a reject state.



Example 3.6 Consider applying the preceding algorithm to build an NFA for $0+10^*$. Start by building NFAs for the 0, the 1, and the 0^*, then combine the latter two, and finally merge the pieces.



Usually the resulting NFA can be easily simplified, as it can in this case.

3.6 Conversion from NFA to DFA

We show next how to convert from an NFA to a DFA. This algorithm is called the **subset construction.** For this, we need to re-examine the workings of an NFA.

Think of how you would determine whether an NFA accepts a given string. The simplest idea is to try systematically all possible transitions. This certainly works, but is not very efficient and takes time exponential in the length of the input.

A better idea is to at each step keep track of the **set of states** the NFA could be in. The key point is that you can determine the set at one step if you know (only) the set at the previous step.

Example 3.7

Consider the behavior of the NFA from Example 3.1 on the string 10100. At the start it is in state A. After reading the 1, the machine is in state A or state C. What happens when the 0 is read? If the machine is in state A, it can choose to go to state A or B. If the machine is in state C it dies. So after the second symbol, the set of states the machine can be in is $\{A, B\}$. Then, when it reads the third symbol, if in state A it goes to $\{A, C\}$, but if in B it dies.

Thus, if we are simulating the machine, it is sufficient to keep track of the set of states the machine could be in. The machine would proceed:

$$\{A\} \xrightarrow{1} \{A, C\} \xrightarrow{0} \{A, B\} \xrightarrow{1} \{A, C\} \xrightarrow{0} \{A, B\} \xrightarrow{0} \{A, B, D\}$$

Does the machine accept 10100? Yes: it can be in an accept state (D) after reading the final symbol.

But there is only a finite number of sets of states. So, the conversion process is to start with the start state, and then to use a systematic process (for example, a breadth-first search) to discover the rest of the diagram. Note that some sets of states might never occur.

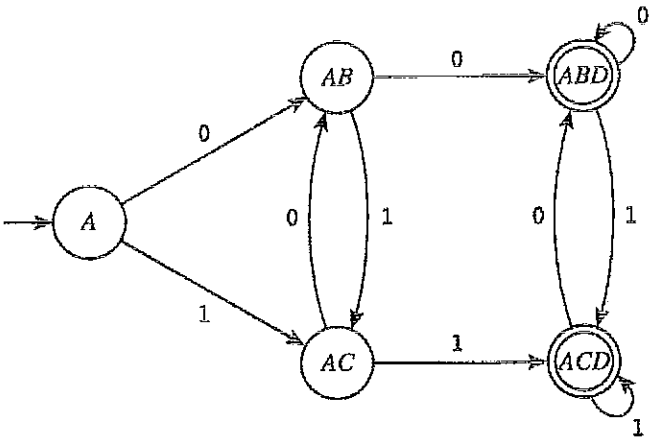
Conversion from NFA (without ε-transitions) to DFA

0. Each state is given by a set of states from the original.
1. Start state is labeled $\{q_0\}$ where q_0 was original start state.
2. While (some state of DFA is missing a transition) do:
   compute the transition by combining the possibilities for each symbol in the set.
3. Make into accept state any set that contains at least one original accept state.

Example 3.8

The following DFA is the result of applying the preceding algorithm to the NFA from Example 3.1.

For example, consider the state $\{A, B, D\}$. On a 1, the NFA, if in state A, can go to states A or C, if in state B dies, and if in state D stays in state D. Thus, on a 1 the DFA goes from $\{A, B, D\}$ to $\{A, C, D\}$. Both of these are accept states because they contain D.



Note that the subset construction does not necessarily give the smallest DFA that does the job. For example, a DFA with fewer states than the one in the previous example was given in Exercise 1.18.

If there are ε-transitions, then you have to adjust the process slightly. Do the following.

Conversion from NFA (with ε-transitions) to DFA

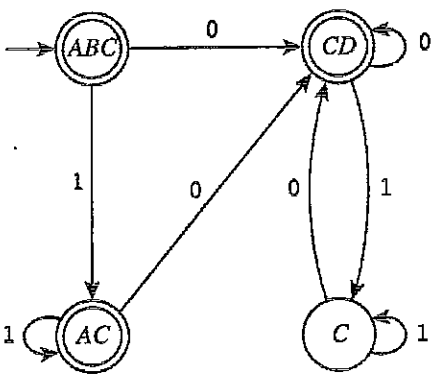
As before, except:

1. The start state becomes the old start state and every state reachable from there by ε-transitions.
2. When one calculates the states reachable from a state, one includes all states reachable by ε-transitions **after** the destination state.

Example 3.9

This is the result of applying the preceding algorithm to the NFA from Example 3.4.

The start state consists of A and the two states you can reach by ε-transitions; the start state is thus $\{A, B, C\}$.



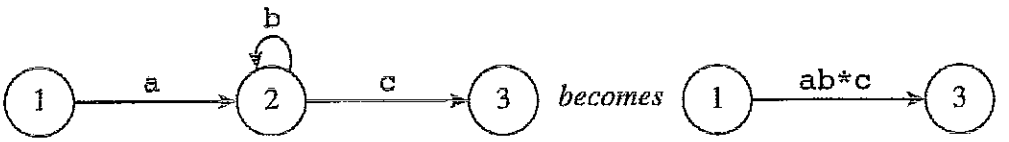
3.7 Conversion from FA to RE

Finally, we show how to convert from an FA to an RE. One way to do this is to generalize even further the notion of an FA. In a **generalized FA** (GFA), each transition is given by an RE. We restrict our attention to the following case:

> *There is a unique accept state, there is no transition out of the accept state, and no transition into the start state (not even a loop).*

Using ε-transitions, you may easily convert any NFA into this form. (How? Left as an exercise.)

What you do now is to build a series of GFAs. At each step, one state other than the start or accept state is removed; the removed state (and its transitions) is replaced by transitions that have the same effect. In particular, if there is a transition a from state 1 to state 2, a transition b from state 2 to state 2, and a transition c from state 2 to state 3, you can achieve the same effect by a transition ab*c from state 1 to state 3.



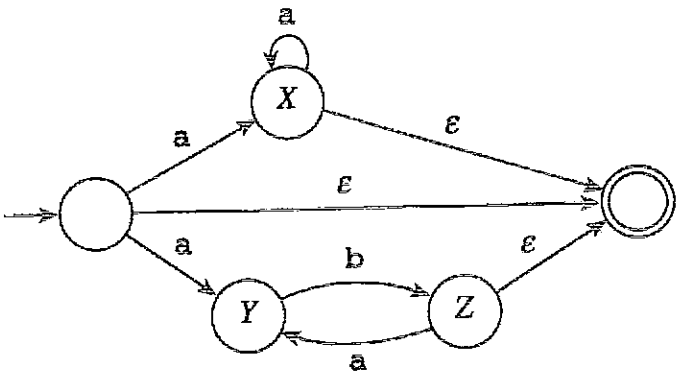
Of course, you need to be convinced that the language accepted by the machine doesn't change. Note, too, that two transitions joining the same pair of states can be merged using the OR of the REs.

Repeat this process until only two states remain; the start and accept states. The label on the single transition joining the two states is a regular expression that corresponds to the original NFA. This process can be summarized as follows:

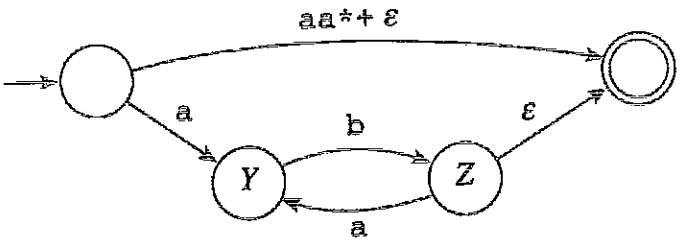
Conversion from FA to RE

0. Convert to FA of the right form (no arc into start, no arc out of unique accept).
1. While (more than two states) do
   remove a state and replace by appropriate transitions.
2. Read RE off the remaining transition.

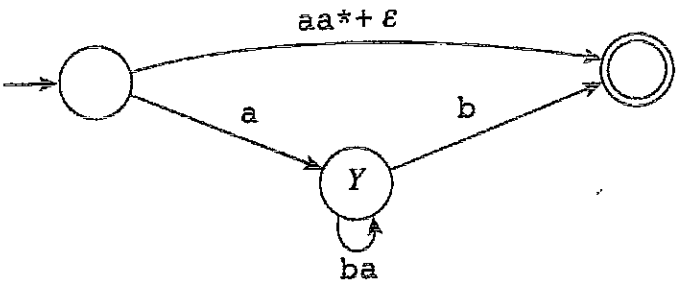
Example 3.10 Here is the NFA of Example 3.3 adjusted to have a unique accept state.



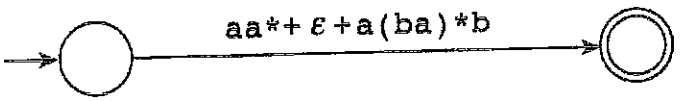
If we eliminate state X, we get



If we eliminate state Z, we get



If we eliminate state Y, we get



For You to Do!

2. Convert the NFA from Example 3.2 to a DFA using the subset construction.
3. Convert the DFA from Example 3.9 to an RE using the preceding method.

EXERCISES

3.1 Give an NFA for the language of the RE $\mathtt{a}^*\mathtt{b}+\mathtt{b}^*\mathtt{a}$.

3.2 Give an NFA for the set of all binary strings that have 0 as the second-to-last symbol.

3.3 Give an FA for the set of strings with alphabet $\{\mathtt{a},\mathtt{b}\}$ that contain both or neither aa and bb as substrings.

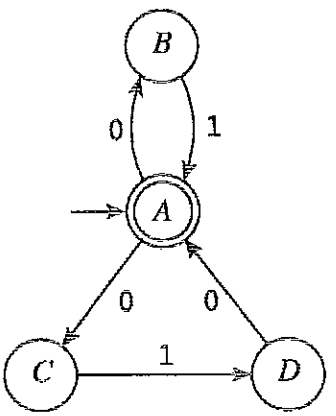
★3.4 Give an FA for the set of strings with alphabet $\{\mathtt{a},\mathtt{b},\mathtt{c}\}$ that have a substring of length 3 containing each of the symbols.

3.5 Show that every finite language has an FA.

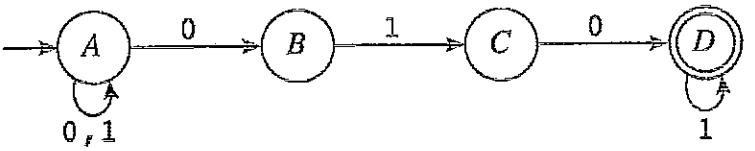
3.6 Show how to modify an NFA to have a unique accept state with no transition ending at the start state and no transition starting at the accept state.

3.7 If M is a DFA accepting language B, then exchanging the accept and reject states gives a new DFA accepting the complement of B. Does this work for an NFA? Discuss.

★3.8 For the following NFA, use the subset construction to produce an equivalent DFA.



3.9 For the following NFA, use the subset construction to produce an equivalent DFA.



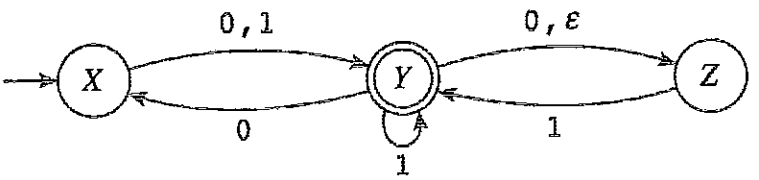
3.10 Consider the RE $(0+01^*)^*$.

- **a)** By following (the spirit of) the text algorithm, produce an equivalent NFA.
- **b)** Describe in English the language of this RE.

3.11 Provide an algorithm to tell if the language is infinite if the input is

- **a)** an RE
- **b)** an NFA

* **3.12** Use the subset construction to convert the following NFA to a DFA.



3.13 Use the GFA algorithm to convert the FA of Example 3.9 to an RE.

3.14 Use the GFA algorithm to convert the final FA of Chapter 1 (Exercise 1.19) to an RE.

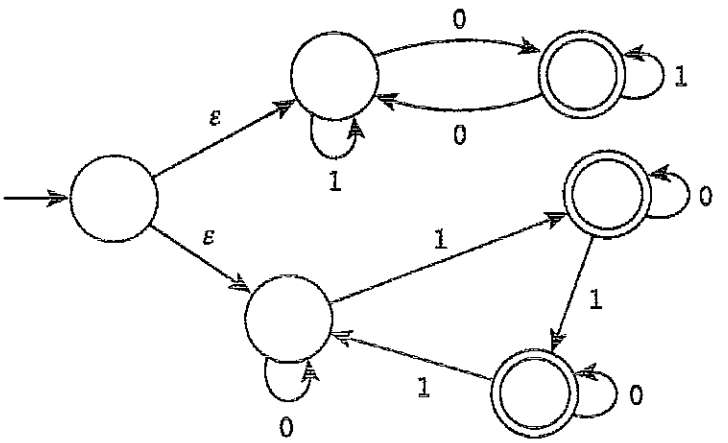
3.15 Discuss the growth in the conversions. If the original object has size n, how large (very roughly) can the new object be?

- **a)** Conversion from RE to NFA
- **b)** Conversion from NFA to DFA
- Ⓗ **c)** Conversion from FA to RE

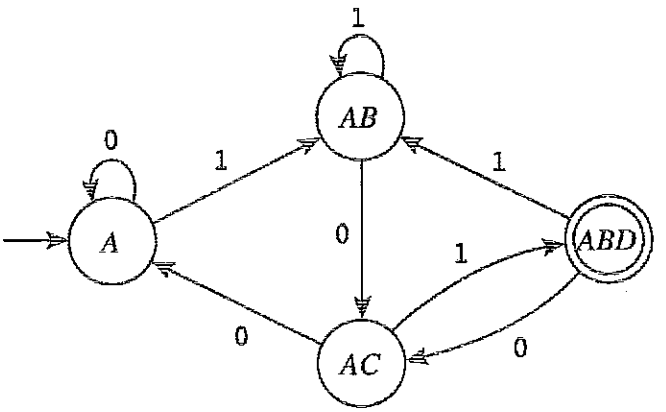
3.16 (TERM PAPER) "Nondeterminism and randomness. Real or imaginary?"

"For You to Do" Exercise Solutions

1.



2.



3. $(0+11^*0)(0+11^*0)^*+\varepsilon+11^*$

chapter

4

Properties of Regular Languages

There are severe limits to the type of recognition problems that can be handled with an FA. But to actually prove that a language is not regular requires some ideas; these are formalized in the notions of distinguishable strings and the infamous Pumping Lemma. We start, however, with some closure properties of the set of regular languages.

4.1 Closure Properties

A set is **closed** under an operation if applying that operation to any members of the set yields a member of the set. For example, the set of integers is closed under multiplication but not under division.

Fact	The set of regular languages is closed under each of the Kleene operations (union, concatenation, and star).

What does this mean? Assume that L_1 and L_2 are regular languages. Then the result says that each of $L_1 \cup L_2$, $L_1 L_2$, and L_1^* is a regular language.

Proof

The easiest way to proceed is to show that the REs for L_1 and L_2 can be combined or adjusted to form the RE for the combination language. For example, the RE for $L_1 L_2$ is obtained by simply writing down the RE for L_1 followed by the RE for L_2. The remaining cases are left as an exercise. ♦

It is important to remember that this question is a property about sets of languages, not about individual strings.

Fact	The set of regular languages is (a) closed under complementation, and (b) closed under intersection.

PROOF

(a) The statement says that if L is a regular language, then so is its complement. The complement of a language, written $\overline{L}$, is its complement as a set: all those strings not in L but with the same alphabet.

To see this fact, look at the deterministic FA for L. To get an FA for $\overline{L}$, simply interchange the accept and reject states. (As an exercise in the previous chapter noted, this interchange does not work with a nondeterministic machine.)

(b) This statement says that if L_1 and L_2 are regular languages, then so is their intersection $L_1 \cap L_2$. To see this, you can use the first part. Then, there is the Venn diagram relation (called de Morgan's law):

$$L_1 \cap L_2 = \overline{(\overline{L_1} \cup \overline{L_2})}$$

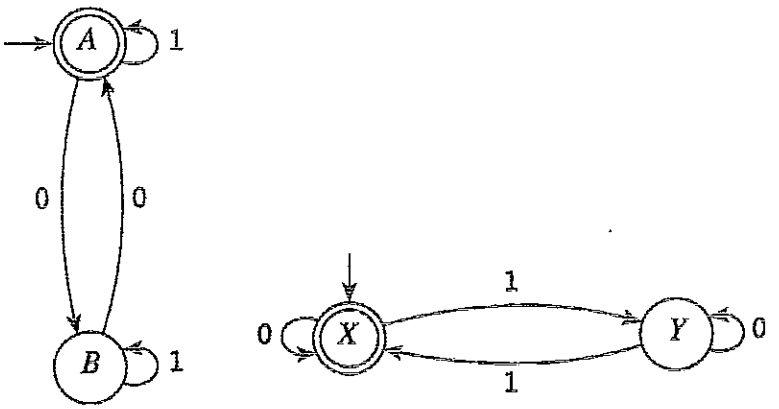
This gives you a procedure to produce an FA for $L_1 \cap L_2$.

The proof of Part (b) actually gives an algorithm (sort of) for producing a DFA for $L_1 \cap L_2$ given the DFAs for L_1 and L_2. Namely, take the DFA for L_1 and DFA for L_2. Interchange accept and reject to yield DFAs for $\overline{L_1}$ and $\overline{L_2}$. Union them (add new start with ε-transitions). Do the subset construction to convert to a deterministic machine. Then interchange accept and reject. Whew!

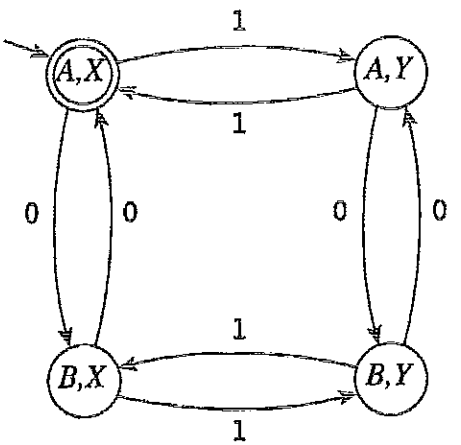
But there is another algorithm for constructing the intersection called the **product** construction. The idea comes from what you would like to do: you would like to run the string simultaneously through both machines. The solution is to keep track of the **pair** of states from each machine.

In particular, the product automaton has one state for every pair of states from the original machines. The start state is the pair of start states, and the accept states are those where *both* states are accept states.

Example 4.4 For example, suppose L_1 is the set of strings with an even number of 0's, and L_2 is the set of strings with an even number of 1's. Then the FAs for these languages both have two states:



And so the FA for the intersection $L_1 \cap L_2$ has four states:



The formal definition of an FA can make this construction look succinct. The **Cartesian product** of two sets A and B, denoted by $A \times B$, is the set $\{(a,b) : a \in A, b \in B\}$. Say L_1 is accepted by automaton M_1 with 5-tuple $(Q_1, \Sigma, q_1, T_1, \delta_1)$ and L_2 is accepted by automaton M_2 with 5-tuple $(Q_2, \Sigma, q_2, T_2, \delta_2)$. Then $L_1 \cap L_2$ is accepted by the automaton $(Q_1 \times Q_2, \Sigma, (q_1, q_2), T_1 \times T_2, \delta)$ where $\delta((r,s), \mathrm{x}) = (\delta_1(r, \mathrm{x}), \delta_2(s, \mathrm{x}))$.

4.2 Distinguishable Strings

Of course, questions about closure would be meaningless without languages that are not regular, which we call the **nonregular** languages.

Example 4.2 Consider the string 0^n1^n; this notation means n 0's followed by n 1's. Define the set B by

$$B = \{0^n1^n : n \geq 0\} = \{\varepsilon, 01, 0011, 000111, \ldots\}$$

No FA exists for this language.

An argument is the following. Suppose that the input string is valid in the sense that it is a sequence of 0's followed by a series of 1's. The machine has to **count** the number of 0's: that is, at the end of the sequence of 0's it must be in a state unique to the number of 0's that were read. But the machine has only fixed finite memory, while the input string is arbitrarily long, and the number of 0's can be arbitrarily large. So the FA has an impossible job.

That is, B is nonregular.

The point is that an FA has only a fixed amount of memory: the information contained in its state. An FA corresponds to a computer with no external memory, just a fixed set of bits that it can manipulate. Note that you can make the memory as large as you want, but once the FA is built, that is that and the memory cannot be expanded.

To make this more precise and more generally applicable, we need some tools. One idea is the concept of distinguishable strings. We say that two strings x and y are **indistinguishable with respect to** L if for every string z, it holds that $xz \in L$ if and only if $yz \in L$. Otherwise, they are called **distinguishable.** That is, x and y are distinguishable with respect to L if there is some suffix z that can be added such that $xz \in L$ and $yz \notin L$ or vice versa.

Theorem Assume M is an FA accepting language L, and let x and y be distinguishable strings with respect to L. Then M must be in a different state after reading x than after reading y.

Proof Suppose the strings x and y put M in the same state. Then for any string z the strings xz and yz put M in the same state. So, if one of xz and yz accepts and one rejects, we have a problem. The only solution is that x and y must put M in different states: the machine must remember which of x or y it has read.

A set of strings is called **pairwise distinguishable** if every pair of strings in it are distinguishable. By the preceding logic, there must be a different state for each string in the set:

Corollary If $\mathcal{D}_L$ is a set of pairwise distinguishable strings with respect to L, then any FA for L has at least $|\mathcal{D}_L|$ states. In particular, if $\mathcal{D}_L$ is infinite then L is not regular.

Example 4.2 (continued) So, back to the language $B = \{0^n 1^n : n \geq 0\}$. We claim that the set $\mathcal{D}_B = \{0^j : j \geq 0\}$ is pairwise distinguishable. Well, take any two strings in $\mathcal{D}_B$; say 0^j and $0^{j'}$ with $j \neq j'$. Then appending 1^j to the first produces a string in B, while appending 1^j to the second produces a string not in B; that is, 0^j and $0^{j'}$ are distinguishable. Because $\mathcal{D}_B$ is infinite, B is not regular.

Example 4.3 Consider the language of Example 4.1. The set $\{\varepsilon, 0, 1, 01\}$ is pairwise distinguishable. So the FA we constructed has the fewest states possible.

We have shown that if there is an infinite set of distinguishable strings for a language, then that language is not regular. There is actually a converse to this; that is, if the language is not regular, then there is guaranteed to be such an infinite set. This result is part of what is called the Myhill–Nerode theorem. In fact, there is a connection between the smallest number of states of an FA and the largest set of pairwise distinguishable strings. Some of the ideas and results are discussed in the exercises.

4.3 The Pumping Lemma

The Pumping Lemma says that every regular language has a certain repetitiveness about it. The lemma can be a bit daunting at first sight.

We will use the following notation. If v is a string, then $|v|$ denotes the length of v. Further, v^3 denotes three consecutive copies of v, and so on.

Pumping Lemma

Let A be a regular language accepted by a DFA with k states. Then, for any string z in A with at least k symbols, you can find an early internal subsegment that can be pumped. That is, z can be split as uvw where:

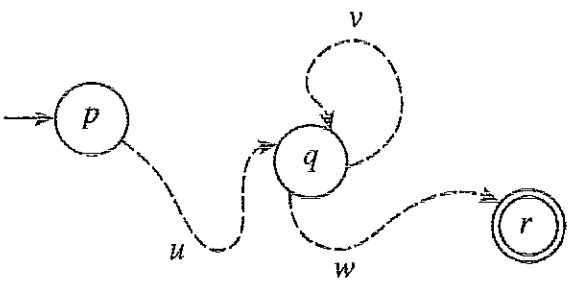
- v is nonempty
- $|uv| \leq k$
- uv^iw is in A for all $i \geq 0$

Proof

Follow the sequence of states around the DFA on the input z. Let q be the first state that occurs a second time. This repeat must happen by the time k symbols are read (because then the automaton has been in $k+1$ states).

Then split the string z as follows:

- String u is the portion up to the first visit to state q.
- String v is the portion between the first and second visit.
- String w is the remainder of the input.



Now the point is that the DFA is in the same state q no matter whether it has read u, uv, uv^2, and so on (it is deterministic). It follows that the DFA is in the same state r no matter whether it has read uw, uvw, or uv^2w. Hence, uv^iw is in A for all $i \geq 0$.

The Pumping Lemma cannot be used to show that a language is regular. Rather, it is used to show that a language is nonregular, by showing that the lemma is contradicted. The art of using the Pumping Lemma is to choose a suitable string z whose pumping causes a contradiction.

Example 4.4 Consider again the language $B = \{ 0^n 1^n : n \geq 0 \}$ from Example 4.2.

Suppose B were regular.
Then B would be accepted by a DFA with k states.
Consider the specific string $z = 0^k 1^k$. This is in B.
Split $z = uvw$ according to the Pumping Lemma.
Then, because $|uv| \leq k$, it follows that v is composed entirely of 0's.
But then uw is not in B (because it has fewer 0's than 1's).
And this is a contradiction of the Pumping Lemma.

It is important to follow the logic once. But the preceding argument can be used as a template. Note that the user has only to find *one* z that cannot be pumped, but must show that there is a problem with *every* possible split.

Example 4.5 Recall that a **palindrome** is a word that reads the same backward as it does forward (such as "level"). Let P be the set of all palindromes for alphabet $\{\mathtt{a}, \mathtt{b}\}$. This language is nonregular. Here is the proof using the Pumping Lemma.

Suppose P were regular. Then it would be accepted by a DFA with, say, k states. Consider the string $z = \mathtt{a}^k\mathtt{b}\mathtt{a}^k$. Split $z = uvw$ according to the Pumping Lemma. Then, because $|uv| \leq k$, it follows that v is always a string of a's. Thus, none of uw, uv^2w, and so forth is in P. This is a contradiction of the Pumping Lemma, and so our supposition is false.

For You to Do! Define E as the language of all binary strings with an equal number of 0's and 1's.

1. Show that E is not regular by finding an infinite set of pairwise distinguishable strings.
2. Show that E is not regular by showing that it contradicts the Pumping Lemma.

EXERCISES

4.1 Show that the set of regular languages is closed under reversal. That is, if L is regular, then so is $\{ x^R : x \in L \}$ where x^R denotes the reversal of string x.

4.2 Give an example to show each of the following for languages L_1 and L_2:

a) If $L_1 \subseteq L_2$ and L_2 is regular, then L_1 can be regular or nonregular.
b) If L_1 and L_2 are nonregular, then $L_1 \cap L_2$ can be regular or nonregular.
c) If L_1 and L_2 are nonregular, then $L_1 \cup L_2$ can be regular or nonregular.

4.3 Suppose language L is accepted by FA M. Let L^E be the subset of L consisting of those strings in L of even length. Show how to convert M to an FA for L^E.

★ **4.4** Show that regularity is closed under prefixes. That is, if L is regular, then so is $\{ x : \text{there is } y \text{ such that } xy \in L \}$.

4.5 Let P_m be the set of all binary strings with a 1 in the mth to last symbol (and at least m symbols). For example, P_1 is the set of all strings ending with a 1. Describe an NFA that accepts P_m and uses $m+1$ states.

4.6 (Continuation of the previous question.)

a) Explain why it follows that there is a DFA that accepts P_m that has 2^{m+1} states.
b) Show that there is a DFA that accepts P_m that has 2^m states.
c) Prove that you need at least 2^m states for a DFA for P_m.

4.7 In a spell checker, it is useful to check whether the given word is one symbol away from a word in the dictionary. For a language L, define L' to the set of all strings obtainable by altering at most one symbol in a string of L. For example, if L is CAT, DOG, then L' is AAT, BAT, CAT, ...,ZAT,...,AOG,...,DOZ. Show how to convert an FA for L into one for L'.

★ **4.8** Prove that the set of all strings of a and b with more a's than b's is nonregular.

4.9 Consider the set of all strings with the alphabet {#} with length a perfect square. Show that this language is nonregular.

4.10 Consider the set of all strings with the alphabet {#} with length a prime number. Show that this language is nonregular.

4.11 Show that $\{ x\#x : x \in \{0,1\}^* \}$ is nonregular. (The hash mark/pound sign is a special symbol that should only occur in the middle of the input string.)

★ **4.12** For each of the following languages, state whether it is regular or not. If not, give a proof that it is nonregular.

a) $\{ (\mathtt{ab})^n : n > 100 \}$
b) $\{ (\mathtt{ab})^n : n < 100 \}$
c) The set of binary strings with both the number of 0's and the number of 1's divisible by 100.

4.13 For each of the following languages, state whether it is regular or not. If not, give a proof that it is nonregular.

a) The set of binary strings with equal number of occurrences of the substrings 01 and 10.
b) The set of binary nonpalindromes.
c) $\{ a^{2^n} : n \geq 0 \}$

4.14 Explain what is wrong with the following "proof" that the language L of the RE a^*b^* is nonregular.

> *Suppose L were regular. Then it would be accepted by a DFA with, say, k states. Consider the string $z = 0^k 1^k$. Split $z = uvw$ with $v = 01$. Then uv^2w is not in L. This is a contradiction of the Pumping Lemma, and so our supposition is false.*

4.15 Consider the language $B = \{ 0^n 1^n : n \geq 0 \}$. Show that a subset of B is regular if and only if it is finite.

★ **4.16** Show that for any fixed m and n, the unary language $\{ \#^{m+ni} : i \geq 0 \}$ is regular.

4.17 (Continuation of the previous question.) Call a unary language an **arithmetic progression** if it is the set $\{ \#^{m+ni} : i \geq 0 \}$ for some m and n.

a) Show that if a unary language is the union of a finite set and a finite number of arithmetic progressions, then it is regular.
b) Show that if a unary language is regular, then it is the union of a finite set and a finite number of arithmetic progressions.

4.18 Let us use the notation $x \equiv_L y$ to mean that strings x and y are indistinguishable with respect to language L.

a) Show that $\equiv_L$ is an **equivalence relation**; that is, for all strings x, y, z, the following hold:
   (i) $x \equiv_L x$
   (ii) If $x \equiv_L y$, then $y \equiv_L x$
   (iii) If $x \equiv_L y$ and $y \equiv_L z$, then $x \equiv_L z$.
b) Show that $\equiv_L$ is a **right congruence**; that is, for all strings x, y, z, the following holds: If $x \equiv_L y$, then $xz \equiv_L yz$.

4.19 (Continuation of previous question.) The equivalence relation $\equiv_L$ partitions the set Σ^* of all possible strings into classes $C_1, C_2, \ldots$ where the strings in each class are pairwise indistinguishable.

a) Show that if $\equiv_L$ partitions Σ^* into an infinite number of classes, then there is an infinite set of pairwise distinguishable strings for L.
b) Show that if there is an infinite set of pairwise distinguishable strings for L, then $\equiv_L$ partitions Σ^* into an infinite number of classes.
c) Show that if $\equiv_L$ partitions Σ^* into a finite number of classes, then there is an FA for L.

d) Show that if there is an FA for L, then $\equiv_L$ partitions Σ^* into a finite number of classes.

4.30 Convince your grandmother that there is no FA that accepts the language of binary strings with an equal number of 0's and 1's.

"For You to Do" Exercise Solutions

1. Let $\mathcal{D}_E$ be the set of all strings containing only 0's. Then for $i \neq j$ the strings 0^i and 0^j are distinguishable with respect to E, since $0^i 1^i \in E$ but $0^j 1^i \notin E$. The set $\mathcal{D}_E$ is infinite, and so E is nonregular.
2. Suppose E were regular. Then it would be accepted by a DFA with, say, k states. Consider the string $z = 0^k 1^k$. Split $z = uvw$ according to the Pumping Lemma. Then since $|uv| \leq k$, it follows that v is always a string of 0's. Thus none of uw, uv^2w, etc. is in E. This is a contradiction of the Pumping Lemma.

Chapter 5

Applications of Finite Automata

So we have seen some of the theory of finite automata. They and their ideas are surprisingly common in computing. In this brief chapter, we discuss some of the places that they are used.

5.1 String Processing

Consider problems involving strings. A standard task is to find all occurrences of a short string, called the *pattern string*, within a long string, called the *text string*. For example, if the pattern is `ana` and the text is `bananarama's ana`, then there are three occurrences of the pattern. Apart from use in editors, this type of question also arises in biology: the text string is the organism's DNA sequence encoded as a string, and the pattern string some DNA sequence of interest.

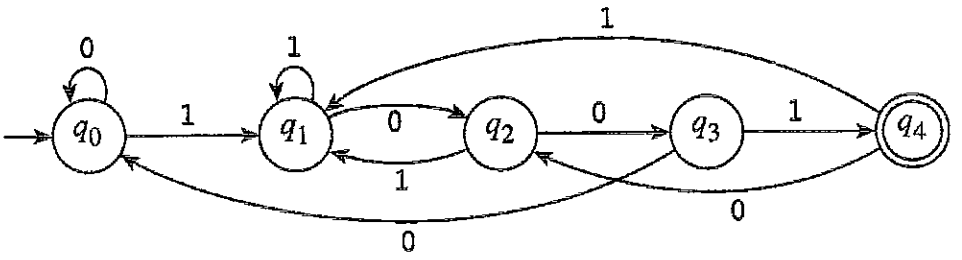
The naive algorithm considers every possible letter of the text as the starting point of the pattern and takes time proportional to the product of the two lengths. The advantage of this approach is its simplicity.

A simple improvement is to use an FA to process the text; in particular, *the DFA for all strings that end with the specified pattern string.* This choice enables you to examine each symbol of the text only once. This is the basis of the Knuth-Morris-Pratt algorithm; they also showed how to build the DFA quickly. The end result is an algorithm that runs in time proportional to the length of the text.

Example 5.1

Suppose you want to know all occurrences of the pattern 1001 in a text string. The approach is to construct the DFA for all strings ending in 1001.

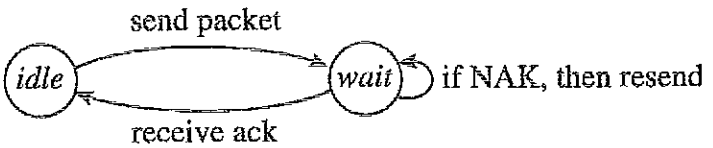
Then, feed the text string through this one symbol at a time. Every time the accept state q_4 is reached, the current position in the text string is printed out.



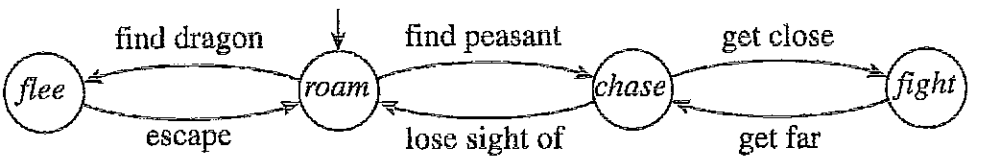
5.2 Finite-State Machines

Finite automata are also used in networks. For example, a communications device is often designed to be in a fixed number of states and to respond to a fixed number of events. For each event, the device's program tells it what action to perform and which state to change to. Here, events such as "receive message" or "timer expires" replace the symbols on the transitions.

The result is usually called a **finite-state machine**: an FA together with actions on the arcs. Here is a trivial example for a communication link:



Finite-state machines are also used in specifying the behavior of computer-generated characters, called **bots**, in a game. You get pictures like the following:



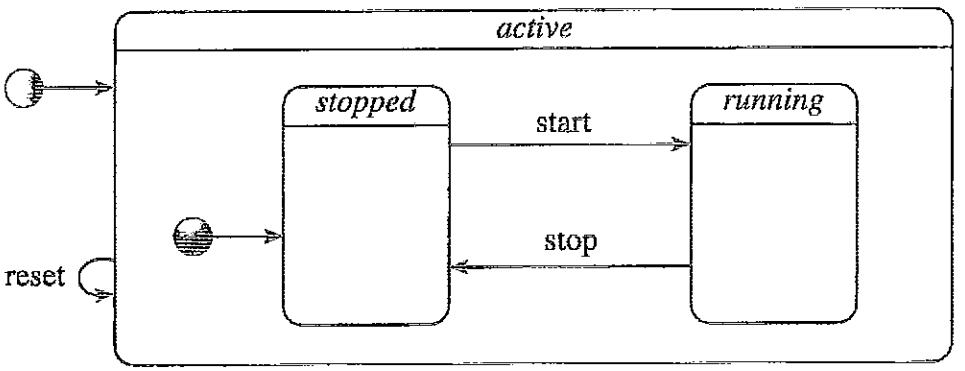
Although the finite-state approach sounds simplistic, and certainly does restrict the range of behaviors of a bot, the advantage is that it's easy to use. Indeed, you can use a metalanguage to specify the diagram and use automated tools to convert to the code.

5.3 Statecharts

Many tasks can be broken up into a set of states and actions to take. For example, the action of dialing on a land-line phone starts with the state of hearing the dial tone. Then the person moves to the state of dialing a number. And so on. **Statecharts** provide a notation for specifying the operation of such processes. These were introduced by Harel in 1987 and are now part of the Unified Modeling Language (UML). Statecharts extend FA diagrams with possibilities such as concurrency, exclusive-or, and hierarchical structures with refinement and superstates.

Example 5.2

Here might be a simplified statechart for a stopwatch.



Apart from clarifying the design, the statechart is also useful in code generation in that there are programs that automatically convert the statechart into a program shell.

5.4 Lexical Analysis

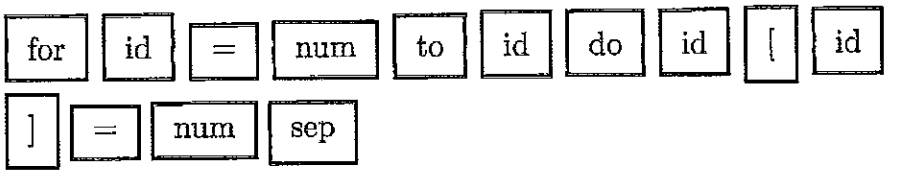
The first step in compiling a Java or C program is called **lexical analysis**. This process isolates keywords, identifiers, operators, and so forth, while eliminating symbols such as comments that are irrelevant for future processing. The input to a lexical analyzer is thus the source code as a string, and the output is a sequence of units called tokens.

A **token** is a category, for example, "identifier" or "relation operator." The **lexeme** is the specific instance of the token. Keywords are often separate tokens.

Example 5.3 The code snippet

```
for i = 1 to max do
    x[i] = 0;
```

might have the token sequence



A programming language is not given by a regular expression. But there are portions that are regular expressions (e.g., numeric constants), and these constitute tokens. For example, the reserved word then and variable names can be described by the following extended regular expressions:

token	*RE*
then	then
variable name	[a-zA-Z] [a-zA-Z0-9]*

where the RE for the variable name says it is any string of alphanumeric characters that starts with a letter.

The lexical analyzer replaces each token by a single value or lexeme. In principle, it goes through the source string one symbol at a time. But in practice it must read ahead to see where the current token ends. If then is read, it's not clear until the next symbol is read if this is a reserved word or an identifier, such as thence. So the lexical analyzer must be able to back up if it reads the next symbol and that's a new token. Also, it is implicit in the definition of the next token that it is the *longest* initial segment that fits one of the token definitions.

As each token is identified, there may be appropriate actions to take. For example, when it identifies a number, the lexical analyzer should also calculate the value of the number. One way to do this is to modify the FA to calculate the value as it goes. The lexical analyzer also keeps track of the variables that have been declared through what is called the symbol table. For example, in strongly typed languages such as Java and C, every variable must be declared before use.

There is a utility distributed with Unix called Lex. It is a program that produces C code for a lexical analyzer. The user supplies the definitions of the tokens and the actions that each token should cause. Approaching lexical analysis in this way has its advantages. The input file to Lex is easily

modified if changes are to be made. It is also easier to convince yourself of the correctness of the Lex input than if you wrote a lexical analyzer from scratch.

EXERCISES

5.1 Draw a DFA for all binary strings ending in `10110`.

5.2 Sketch a statechart or a finite-state machine that would represent the operation of a telephone.

5.3 Draw a DFA that accepts all strings that represent decimal numbers (such as `3.1415`). Add actions to the transitions so that it calculates the value of the number as it reads the string.

5.4 (TERM PAPER) Take one of the applications mentioned here and investigate further.

SUMMARY

An alphabet is a set of symbols. A string is a finite sequence of symbols drawn from some alphabet. A language is any set of strings. The empty string is denoted ε.

A finite automaton (FA) is a device that recognizes a language. It has finite memory and an input tape; each input symbol that is read causes the machine to update its state based on its current state and the symbol read. The machine accepts the input if it is in an accept state at the end of the string; otherwise, the input is rejected.

A regular expression (RE) is built up from individual symbols using the three Kleene operators: union ($+$), concatenation, and star (*). The star of a language is obtained by all possible ways of concatenating strings of the language, repeats allowed; the empty string is always in the star of a language.

A nondeterministic finite automaton (NFA) can have zero, one, or multiple transitions corresponding to a particular symbol. It is defined to accept the input if there exists some choice of transitions that cause the machine to end up in an accept state. Nondeterminism can also be viewed as a tree, or as a "guess-and-verify" concept. You can also have ε-transitions, where the NFA can change state without consuming an input symbol.

Kleene's theorem says that the following are equivalent for a language: there is an FA for it; there is an NFA for it; and there is an RE for it. The proof provides an algorithm to convert from one form to another; the conversion from NFA to DFA is the subset construction.

A regular language is one that has an FA or an RE. Regular languages are closed under union, concatenation, star, and complementation. To show that a language is nonregular, you can show that there is an infinite set of pairwise distinguishable strings, or use the Pumping Lemma and show that there is some string that cannot be pumped.

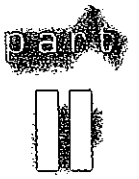
Applications of finite automata include string-matching algorithms, network protocols, and lexical analyzers.

INTERLUDE: JFLAP

Many of the models in this text can be demonstrated with an interactive visualization and teaching tool for formal languages titled JFLAP. JFLAP is an acronym that stands for Java Formal Language and Automata Package. JFLAP allows users to create and operate on automata, grammars, L-systems, and regular expressions. It is written in Java, hence it runs on most platforms. The software program is available as a free download at www.jflap.org. Please visit the website to register along with the thousands of other students that currently enjoy this interactive tool.

The JFLAP package implements several components. For example, it handles construction of FAs and other models. It also handles the conversion and simulation algorithms discussed in this text, not just the answers. JFLAP will walk the user through each step.

JFLAP allows for and encourages experimentation. One could change one piece and have JFLAP recalculate the result. There are additional models offered with JFLAP. For example, check out JFLAPs L-systems for its vivid pictures.



Context-Free Languages

In the first part of the book, we investigated regular languages. We saw a model computer that recognizes regular languages and a regular expression that generates regular languages. Indeed, several interesting languages are regular. But we saw that some languages that are simple to describe do not fall into this class; you need more power.

So we go in search of a larger family of languages. We now enhance the automaton by giving it some external storage. This storage will initially be in the form of a stack. This is perhaps not the most obvious choice, but it has proved over time to be the most appropriate next level. We shall see that this allows some nonregular languages to be recognized, but there are still simple-looking languages that fall outside this class.

We start, however, with a mechanism to generate strings, called a grammar. Grammars were originally invented to describe features of human spoken languages. They have proved useful as a way to describe computer languages, and even as a programming tool. Our main focus here is on a special type of grammar, known as a context-free grammar.

Language is a form of human reason, which has its internal logic of which man knows nothing. —**Claude Levi-Strauss**

It is of interest to note that while some dolphins are reported to have learned English—up to fifty words used in correct context—no human being has been reported to have learned dolphinese. —**Carl Sagan**

I don't want to talk grammar, I want to talk like a lady. —**George Bernard Shaw, Pygmalion**

Colorless green ideas sleep furiously. —**Noam Chomsky**

Chapter 6

Context-Free Grammars

A grammar for a computer language is much like that for a natural language: a set of rules for putting things together. Each grammar corresponds to a language. Our focus is on a special type of grammar, called a context-free grammar.

6.1 Productions

This is an example of a context-free grammar:

Example 6.1

$S \to 0S1$
$S \to \varepsilon$

A grammar consists of

- A set of **variables** (also called nonterminals)
- A set of **terminals** (from the alphabet)
- A list of **productions** (also called rules)

In the preceding example, S is the only variable. The terminals are 0 and 1. There are two productions. It is customary to use upper-case letters for variables.

How does a grammar work? A production allows you to take a string containing a variable and replace the variable by the right-hand side of the production. We say that a (finite) string w, consisting of terminals, is generated by the grammar if, starting with the start variable S, you can

apply productions and end up with that string. The sequence of strings so obtained is called a **derivation** of w. We will see later other versions of a grammar; this version is called a **context-free grammar** (CFG). A language is **context-free** if it is generated by a CFG.

Example 6.1 (continued)

The string 0011 is in the language generated. The derivation is:

$$S \Longrightarrow 0S1 \Longrightarrow 00S11 \Longrightarrow 0011$$

The first two steps use the first production; the final step uses the second production.

What does this language contain? Certainly every string generated has an equal number of 0's and 1's. In fact, it is our nonregular language $\{ 0^n 1^n : n \geq 0 \}$.

Notice that a CFG has a **recursive** flavor about it. In the preceding example, the CFG says that all strings in the language can be built up starting with the empty string and repeatedly wrapping with the symbols 0 and 1. Without recursion, the grammar would only define a finite language.

As for notation, we could also write the preceding CFG as

$$S \rightarrow 0S1 \mid \varepsilon$$

where the vertical bar means *or*.

Example 6.2

We saw earlier that the language P of palindromes with alphabet $\{\mathtt{a}, \mathtt{b}\}$ is nonregular. We can determine a CFG for it by finding a recursive decomposition. If you peel the first and last symbols from a palindrome, what remains is a palindrome; and if you wrap a palindrome with the same symbol in front and in back, then it is still a palindrome. This is the basis for the following CFG:

$$P \rightarrow \mathtt{a}P\mathtt{a} \mid \mathtt{b}P\mathtt{b} \mid \varepsilon$$

Actually, this generates all palindromes of even length. An exercise asks how to adapt this CFG to generate all palindromes.

Mathematicians might write the language of even-length palindromes as $\{ w\, w^R : w \in \Sigma^* \}$.

We can provide a **formal definition** of a context-free grammar. It is a 4-tuple (V, Σ, S, P) where:

- V is a finite set of variables.
- Σ is a finite alphabet of terminals.
- S is the start variable.
- P is the finite set of productions. Each production has the form $V \to (V \cup \Sigma)^*$.

6.2 Further Examples

We start with CFGs for some regular languages.

Example 6.3

A CFG for all binary strings with an even number of 0's.

Well, how to decompose such a string? If the first symbol is a 1, then an even number of 0's remains. If the first symbol is a 0, then go to the next 0; what remains after that symbol is again a string with an even number of 0's. This yields the following CFG:

$$S \to 1S \mid 0A0S \mid \varepsilon$$
$$A \to 1A \mid \varepsilon$$

But a language can have more than one grammar. For example, when the first symbol is a 0, you might also observe that what remains has an odd number of 0's. If we use T to generate all binary strings with an odd number of 0's, we get the following CFG:

$$S \to 1S \mid 0T \mid \varepsilon$$
$$T \to 1T \mid 0S$$

Example 6.4

A CFG for the regular language corresponding to the RE 00^*11^*.

The idea is that the language is the concatenation of two languages: the first language is all strings of 0's, and the second is all strings of 1's.

$$S \to CD$$
$$C \to 0C \mid 0$$
$$D \to 1D \mid 1$$

The next example is the complement of the language from the previous example. There is no obvious way to convert a grammar to its complement. The goal here is a CFG for all strings *not* of the form $0^i 1^j$ where $i, j > 0$. One approach is to partition the set of strings into the three failures:

- String not of the right form: somewhere there is a 1 followed by a 0
- Only zeroes
- Only ones

This yields the following CFG.

Example 6.5 **Complement of $\{ 0^i 1^j : i, j > 0 \}$.**

$$S \to A \mid B \mid C$$
$$A \to D10D$$
$$D \to 0D \mid 1D \mid \varepsilon$$
$$B \to 0B \mid 0$$
$$C \to 1C \mid 1$$

The variable A will produce all strings with a 0 and 1 out of order, variable B will produce strings of zeroes, variable C will produce strings of ones, and variable D will produce all strings.

Note that to check that a grammar and a description match, you must check two things: that everything the grammar generates fits the description (sometimes called **consistency**), and that everything in the description is generated by the grammar (sometimes called **completeness**).

Example 6.6 Consider the CFG

$$S \to 0S1S \mid 1S0S \mid \varepsilon$$

The string 011100 is in the language generated by the preceding grammar. One derivation is the following:

$$S \Longrightarrow 0S1S \Longrightarrow 01S \Longrightarrow 011S0S \Longrightarrow 0111S0S0S \Longrightarrow 01110S0S$$
$$\Longrightarrow 011100S \Longrightarrow 011100$$

What does this language contain? Certainly, every string generated has an equal number of 0's and 1's. But can any string with an equal number of 0's and 1's be generated? Yes. Why?

Well, whatever the string starts with, at some point, equality between the numbers of symbols must be reached. The key point is that if the string starts with a 0, then the first time equality is reached is at a 1. Then the portion of the string between the first symbol and this 1 is itself an example of equality, as is the portion after this 1. That is, you can break up the string as $0w1x$, where both strings w and x belong to the language. For example, here is the breakup of 00101101:

$$0\underbrace{0\ 1\ 0\ 1}_{w}1\underbrace{0\ 1}_{x}$$

This argument shows the completeness: every string with an equal number of zeroes and ones is generated.

Grammars originally arose in the study of human languages. For example, here is a primitive grammar that generates sentences as composed of noun and verb phrases:

Example 6.7

$$\begin{aligned} S &\to NP\ VP \\ NP &\to \texttt{the}\ N \\ VP &\to V\ N \\ V &\to \texttt{sings} \mid \texttt{eats} \\ N &\to \texttt{cat} \mid \texttt{song} \mid \texttt{canary} \end{aligned}$$

This generates "the canary sings the song", but also "the song eats the cat".

The above CFG generates all "legal" sentences, not just those that are meaningful.

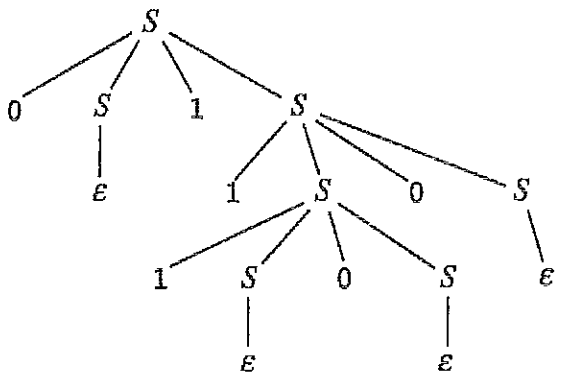
6.3 Derivation Trees and Ambiguity

For strings α and β, we say α **yields** β, written $\alpha \stackrel{*}{\Longrightarrow} \beta$ if it is possible to get from α to β using the productions. A **derivation** of β is the sequence of steps that gets to β. A **leftmost** derivation is where at each stage one replaces the leftmost variable. In Example 6.6, the derivation of 011100 that is given is a leftmost derivation. A **rightmost derivation** is defined similarly.

You can also represent the derivation by a **derivation tree**. The root of the tree is the designated start variable, all internal nodes are labeled with variables, while the leaves are labeled with terminals. The children of a node are labeled from left to right with the right-hand side of the production used. There is a 1-to-1 correspondence between derivation trees, leftmost derivations, and rightmost derivations.

Example 6.8

The derivation tree for the string 011100 in the grammar of Example 6.6 is drawn:



We say that a grammar is **unambiguous** if there is a unique leftmost derivation for each string in the language. Equivalently, for each string in the language there is a unique derivation tree. For example, the grammar in Example 6.6 is ambiguous—but not for the example string. (The string 0101 has two derivation trees—find them.)

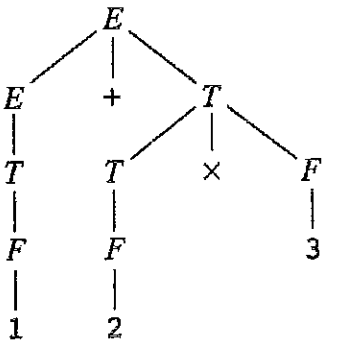
Grammars are used in compilers. A compiler checks that the input file or expression is valid by essentially finding the derivation tree. This derivation tree is then used in finding code for the file or expression. If the grammar is unambiguous, the derivation tree is always unique. This process is discussed in Chapter 10.

Example 6.9

The language of arithmetical expressions using only multiplication and addition can be described as the following with start variable E:

$$E \to E + T \mid T$$
$$T \to T \times F \mid F$$
$$F \to (E) \mid \textit{number}$$

(Think of Expression, Factor, and Term.) This generates expressions such as 1+(3+2)×5 and 1+2×3. The derivation tree for the latter is as follows.



Note that the derivation tree automatically knows that multiplication takes precedence over addition.

6.4 Regular Languages Revisited

What is the relationship between regular languages and context-free languages? The following is probably not surprising.

Theorem	Every regular language is generated by a context-free grammar.

Proof

One way to prove this is to convert an RE to a CFG. This can be achieved recursively. For example, if the overall language is the union of two pieces, you can write $S \to A \mid B$; and if it is the concatenation of two pieces, you can write $S \to CD$. If the overall language is the star of a piece, say, generated by E, then you can write $S \to ES \mid \varepsilon$.

Example 6.10

Consider the RE $(11+00)^*11$. At the top level, it is the *concatenation* of two pieces. The first piece is the *or* of two parts. This yields the following grammar:

$$S \to TU$$
$$U \to 11$$
$$T \to TV$$
$$V \to 00 \mid 11$$

For You to Do!

Give grammars for the following two languages:

1. All binary strings with both an even number of zeroes and an even number of ones.
2. All strings of the form $0^a 1^b 0^c$ such that $a + c = b$. (Hint: It's the concatenation of two simpler languages.)

EXERCISES

6.1 Give a CFG for palindromes that allows odd-length palindromes.

6.2 Construct a CFG for the language of binary strings of the form $0^n 1^{2n}$.

6.3 For each of the following CFGs: (1) determine which of the strings ε, abba, aaaaa, and b is generated, and (2) give an English description of the language.

a) $S \to \mathtt{a}S \mid \mathtt{b}S \mid \mathtt{a} \mid \mathtt{b} \mid \varepsilon$

b) $S \to X\,\mathtt{aaa}X$
$X \to \mathtt{a}X \mid \mathtt{b}X \mid \varepsilon$

c) $S \to \mathtt{aa}S \mid \mathtt{aaa}S \mid \mathtt{a}$

d) $S \to \mathtt{a}X \mid \mathtt{b}S \mid \mathtt{a} \mid \mathtt{b}$
$X \to \mathtt{a}X \mid \mathtt{a}$

★ **6.4** Show that the set of context-free languages is closed under the three Kleene operators.

6.5 Give CFGs for all strings of the form:

a) $10^n 10^n 1$

b) $0^n 1^m 2^k$ with $n = m + k$

c) $0^n 1^m 0^m 1^n$

d) $0^i 1^j 2^k$ with $i = j$ or $i = k$

6.6 Give CFGs for the following languages with alphabet $\{\mathtt{a}, \mathtt{b}\}$:

a) All strings of the form $\mathtt{a}^m\mathtt{b}^n$ with $m \leq n \leq 2m$.

b) All strings such that the middle symbol is a.

6.7 Give a CFG for the complement of $\{0^n 1^n : n \geq 0\}$.

★ **6.8** In a string, a **block** is a substring, all of whose symbols are the same and which cannot be enlarged. For example, 0001100 has *three* blocks. Let L be the language consisting of all binary strings that have (somewhere) two blocks of zeroes of the same length. For example, $011001110 \in L$ and $11101011 \in L$, but $000110011 \notin L$. Give a grammar for L.

Ⓗ **6.9** Construct a CFG for the language of binary strings with an unequal number of 0's and 1's.

6.10 Consider the following CFG with start variable S:

$$S \to BB$$
$$B \to SS \mid \mathtt{c}$$

a) What is the shortest string in the language of the grammar?
b) Draw the derivation tree for the string from (a).
c) Is this grammar ambiguous? Explain.
d) Describe in English the language generated by the grammar.
e) Does this CFG generate a regular language? Explain.

6.11 Describe in English the language generated by the following grammar:

$$S \to 0S1 \mid 1S0 \mid \varepsilon \mid SS$$

Sketch a proof that your answer is correct.

★ **6.12** Explain what language this CFG generates (with start variable S):

$$S \to AB$$
$$A \to 0A1 \mid \varepsilon$$
$$B \to 1B0 \mid \varepsilon$$

Sketch a proof that your answer is correct.

6.13 Give the shortest string generated by the following CFG (with start variable S and alphabet $\{0, 1\}$), and give a derivation tree for that string:

$$S \to ABA \mid SS$$
$$A \to S0 \mid T1T$$
$$B \to S1 \mid 0$$
$$T \to 0$$

6.14 Consider the language L of all strings of a's and b's such that there are more a's than b's. Does the following grammar generate L? Justify your answer.

$$S \to \mathtt{a}S\mathtt{b} \mid \mathtt{b}S\mathtt{a} \mid S\mathtt{ba} \mid S\mathtt{ab} \mid \mathtt{ab}S \mid \mathtt{ba}S \mid \mathtt{a}S \mid S\mathtt{a} \mid \mathtt{a}$$

Ⓗ **6.15** Consider the language D of all strings of a's and b's with twice as many a's as b's.

a) Does the following grammar generate D? Justify your answer.

$$S \to \mathtt{a}S\mathtt{a}S\mathtt{b}S \mid \mathtt{a}S\mathtt{b}S\mathtt{a}S \mid \mathtt{b}S\mathtt{a}S\mathtt{a}S \mid \varepsilon$$

b) Does the following grammar generate D? Justify your answer.

$$S \to SS \mid AB \mid BA \mid \varepsilon$$
$$A \to AS \mid SA \mid \mathtt{a}$$
$$B \to SB \mid BS \mid A\mathtt{b} \mid \mathtt{b}A$$

★ **6.16** Consider the grammar $S \to SS \mid SSS \mid \mathtt{x}$.

a) Draw all derivation trees for xxx.
b) How many derivation trees are there for xxxx? Explain.

6.17 For the CFGs in Exercise 6.3, determine which are ambiguous, and for each of those, find a string that shows that the CFG is ambiguous.

6.18 Let G be a CFG in which every variable occurs on the left-hand side of at most one production. Prove that G is unambiguous.

6.19 Consider a programming language Puny that allows only a few things. These include declaration of `int` variables; assignments where the right-hand side uses only operators, variables, and constants; a `begin` only at the beginning of a program and `end` at the end; all declarations at the start; and all statements ending with a semicolon. There is also a loop construct that can be nested.

a) Derive a grammar.
b) Sketch the parse tree for the following code. (The tree is huge! So draw it in pieces.)

```
begin
  int x; int y;
  while x!=y :
    x = x+1;
    while x!=0 :
       y = y*x + 22;
    endwhile;
  endwhile;
end
```

★ **6.20** Give a CFG for the language R where R is the set of all REs with alphabet $\{\mathtt{a}, \mathtt{b}\}$.

"For You to Do" Exercise Solutions

1. $S \to 0X \mid 1Y \mid \varepsilon$

$X \to 0S \mid 1Z$ (odd zeroes, even ones)

$Y \to 1S \mid 0Z$ (odd ones, even zeroes)

$Z \to 0Y \mid 1X$ (odd ones, odd zeroes)

2. $S \to TU$

$T \to 0T1 \mid \varepsilon$

$U \to 1U0 \mid \varepsilon$

Chapter 7

Pushdown Automata

We consider now a model of a computer that is more powerful than a finite automaton. In this model, we give the FA some memory; in particular, we give it a stack.

7.1 A PDA Has a Stack

A **stack** is a way of storing information on a **last-in, first-out** principle. Everything that is added to a stack is added to the top, and everything that is removed from the stack is removed from the top. There is always only one exposed item. The process of adding an item to the stack is called **pushing**; the item goes on the top. The process of removing an item from the stack is called **popping**; the item removed was the top item.

A **pushdown automaton** (PDA) is like an FA in that there is a fixed finite number of states that it can be. But it also has one unbounded stack for storage. A PDA works by reading the input. When a symbol is read, depending on

a) the state of the machine,
b) the symbol on top of the stack, and
c) the symbol read,

the machine

1. updates its state, and
2. pops or pushes a symbol.

The machine may also pop or push without reading input.

We draw the **program** of a PDA as a **flowchart**. There are five components to this:

- A single **start** state
- A single **halt-and-accept** state
- A **reader** box: this reads one symbol from the input and, depending on the symbol read, goes to a new state (just as in an FA)
- A **pop** box: this pops one symbol from the stack and, depending on the symbol, goes to a new state
- A **push** box: this adds a specific symbol to the stack

Note that the flowchart has no reject state. Rather, if the machine gets to a state and there is no legal continuation, then *we assume the machine halts and rejects the input.*

We also need a **special symbol**: we will use Δ to indicate the end of the input string. We will also use Δ to indicate the result of popping when the stack is empty—you could also think of this as the stack starting with the special symbol Δ on it.

Example 7.1

The language $\{ 0^n 1^n : n > 0 \}$. (See Example 6.1.)

The PDA will use the stack as a counter. The machine starts with an empty stack. As a symbol comes in, the machine checks whether it is a 0. If so, it pushes an x onto the stack. When the first 1 is encountered, the machine enters a new state. In this state, it pops symbols off the stack, one for each input 1. If, during this phase, a 0 is encountered or the stack becomes empty, then the machine rejects the string (because either the string doesn't have the right form or there are too few 1's). The PDA accepts if and only if the stack becomes empty at the same time the end of the string is reached.

The PDA is shown in Figure 7.1. To get a feel for it, walk through this PDA with the strings 000111, 0010, and 011. (Some would prefer to push 0's rather than x's onto the stack.)

Casualness

There are traditional shapes for the different types of functions on the flowchart—diamonds for decisions based on reading or popping, rectangles for pushes, and lozenges for start and accept states—but don't worry too much about that. Also, the empty string often requires very special handling; from now on, however, we will simply *ignore the empty string.*

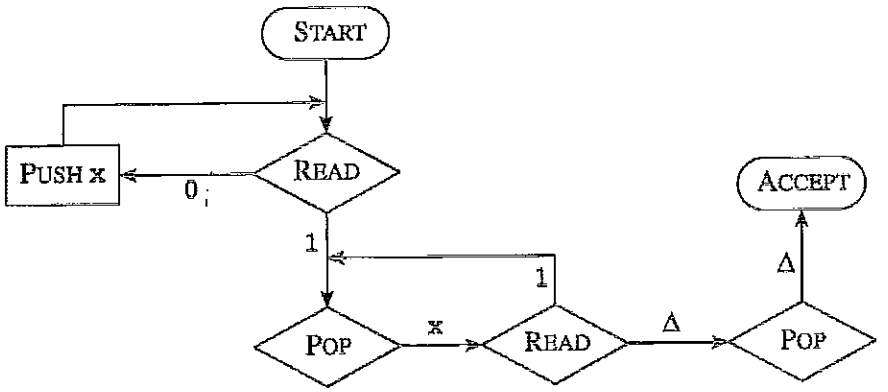


Figure 7.1 A PDA for 0^n1^n

Example 7.2

Balanced brackets.

We consider here strings consisting entirely of left and right brackets. Such a string is called **balanced** if (a) reading from left to right the number of left brackets is always at least the number of right brackets; and (b) the total number of left brackets equals the total number of right brackets. For example,

`(()())()` is balanced; `(()` and `)))(` are not.

In a normal arithmetical expression, the brackets are always balanced.

It is easy to produce a grammar for such strings:

$$S \to (S) \mid SS \mid \varepsilon$$

In the PDA for such strings, each opening bracket (is simply pushed; each closing bracket) causes a matching (to be popped. The PDA is drawn in Figure 7.2.

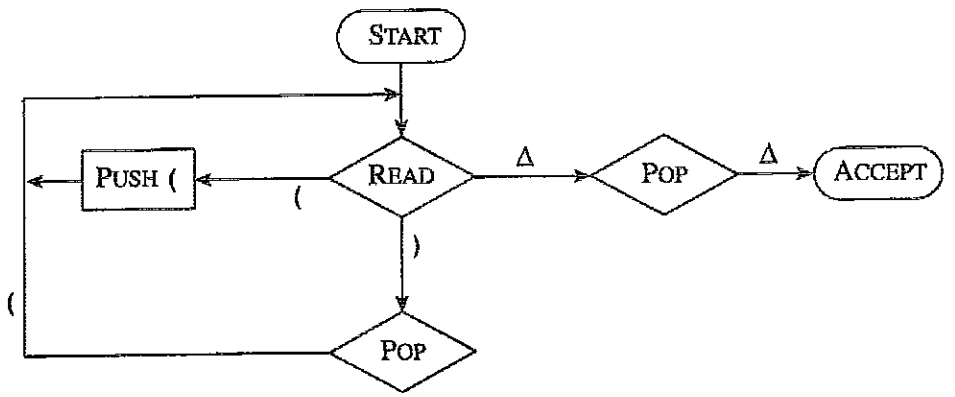


Figure 7.2 A PDA for balanced brackets

7.2 Nondeterminism and Further Examples

What about a PDA for palindromes? What we do is use nondeterminism. In fact, *by definition*, a PDA is nondeterministic. We say that the PDA accepts the input string if **there exists** a sequence of actions that leads to the accept state. There are two ways to depict nondeterminism in the flowchart: there can be two transitions with the same label, or there can be a transition labeled with ε (which does not consume an input symbol).

So what about a PDA for palindromes? Well, you read the input and stack it away merrily. Until suddenly you have this premonition that the middle of the road has been reached. Then, you start popping, each time checking that the input symbol matches the one on the stack. If ever there is a conflict, the machine dies gracefully. If the input ends before the stack is empty, or the stack is empty before the input ends, then again it dies. The machine accepts the string if it reaches the empty stack at the end of the input and never encounters a conflict. This PDA is shown in Figure 7.3.

It can be proven that you need nondeterminism for palindromes.

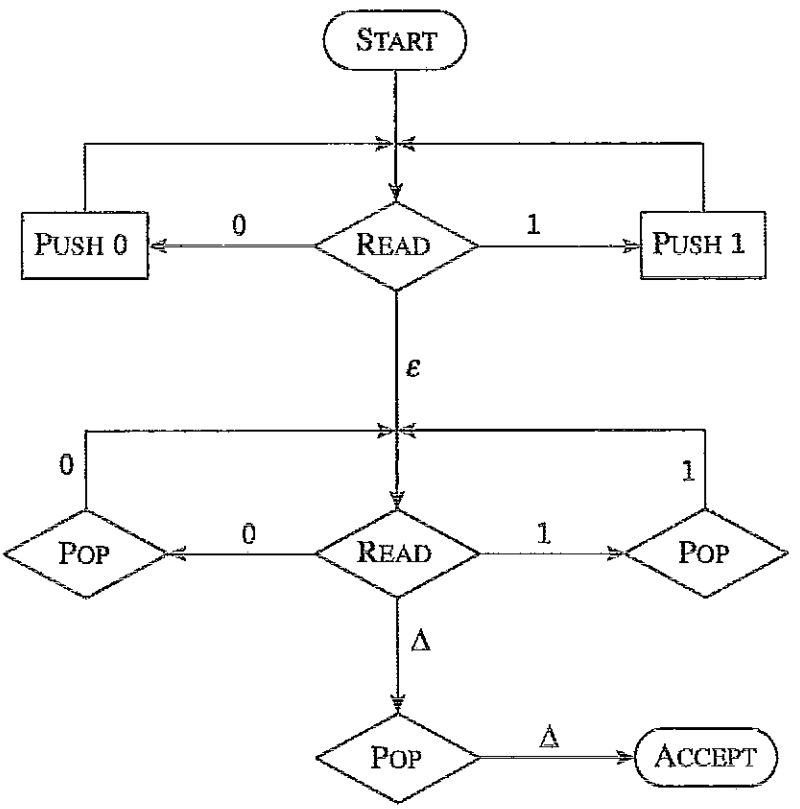


Figure 7.5 A PDA for even-length palindromes

Here is another language that benefits from nondeterminism.

Example 7.3

The language $\{ 0^m 1^n : n \leq m \leq 2n \}$.

The first phase should be clear: you need to count the number of zeroes. Say we use the stack symbol x like we did with $0^n 1^n$. After that, we want to match zeroes with ones. Actually, to make this work, it is sufficient to match each 1 with either one or two x's. You can guess when to do the change over, or you can simply intermingle the matches, allowing the nondeterminism to choose the correct path. The PDA with the former approach is drawn in Figure 7.4.

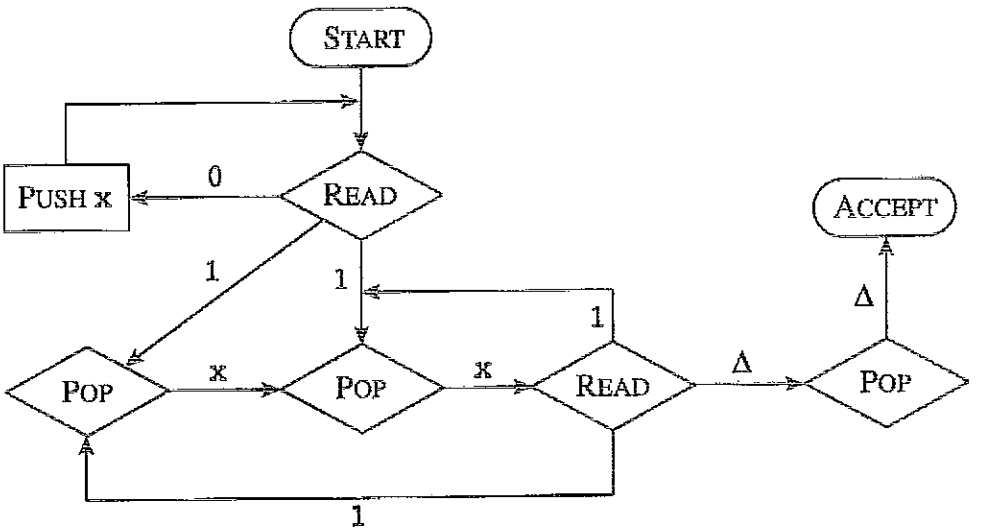


Figure 7.4 A PDA for $\{0^m 1^n : n \leq m \leq 2n\}$

The next PDA is for all binary strings with an **equal number of 0's and 1's**. The idea is to use the stack as a counter, but the details can be a bit complex. It is natural that the stack keeps track of the difference between the number of 0's and 1's so far. The key idea to get a simple PDA is to store 0's on the stack if the 0's are in the majority, and to store 1's if the 1's are in the majority.

The PDA is shown in Figure 7.5. How does it work? Consider the leftmost read state. The invariant is that whenever here, the 0's are in the majority, and the stack has *one less* 0 than the excess (so the stack is empty when the 0's outnumber the 1's by only one). We are at the middle read state whenever there have been equal numbers of the two symbols.

We defer the formal definition of a PDA until the next chapter because there is an alternative representation that is more in line with the formal definition.

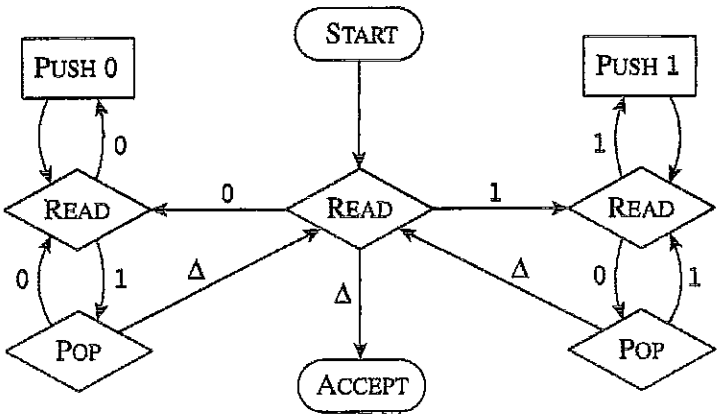


Figure 7.5 A PDA for equal 0's and 1's

7.3 Context-Free Languages

The big theorem (if you hadn't guessed it by now) is as follows.

Theorem	A language is generated by a context-free grammar if and only if it is accepted by a pushdown automaton.

We prove the theorem in the next chapter. It follows that a context-free language is one that is generated by a CFG or accepted by a PDA.

7.4 Applications of PDAs

Pushdown automata are useful in compilers. For example, an arithmetical expression can be first understood using a stack, and then evaluated using a stack, as the following example shows.

Example 7.4 Recall the CFG for arithmetic from Example 6.9. A postorder traversal of the derivation tree provides what is called the **reverse Polish** form of the expression. This can be used to provide code for evaluation using a stack.

For example, if MUL is a function to replace the top two values on the stack by their product, and ADD replaces the top two values on the stack by their sum, an expression evaluator might convert

$$1 + 5 * (3 + 2) + 4$$

into the code

PUSH(1) PUSH(5) PUSH(3) PUSH(2) ADD MUL ADD PUSH(4) ADD

For You to Do!

1. Draw a PDA for the set of all strings of the form $0^a 1^b$ such that $a \geq b$.
2. Draw a PDA for the set of all strings of the form $0^a 1^b 0^c$ such that $a + c = b$.

EXERCISES

7.1 Adapt the PDA from Figure 7.3 to accept odd-length palindromes as well.

7.2 Construct a PDA for the language of strings of the form $0^n 1^{2n}$.

7.3 Construct a PDA for the language of strings of the form $0^n 10^n$.

★ **7.4** Construct a PDA for strings that consist of three parts, where the first part is a binary string, the second part is the symbol #, and the third part is the reverse of the first part. For example, 01001#10010 is in the language.

7.5 Walk through the silly PDA in Figure 7.6 with the strings abba, babab, and baaa. What language does it accept?

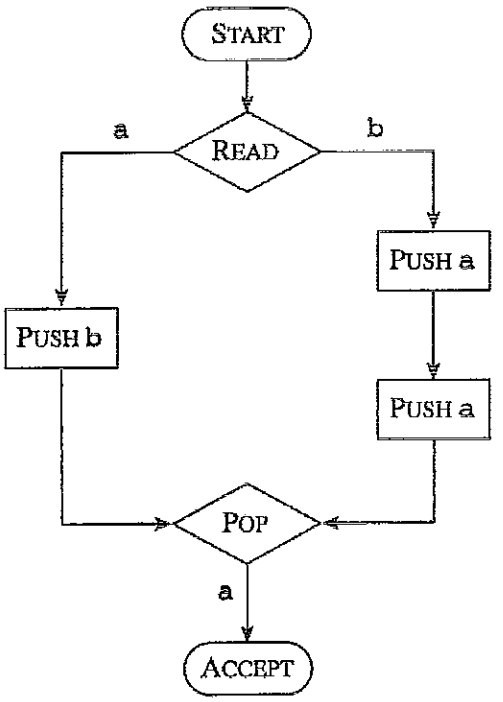


Figure 7.6 A silly PDA

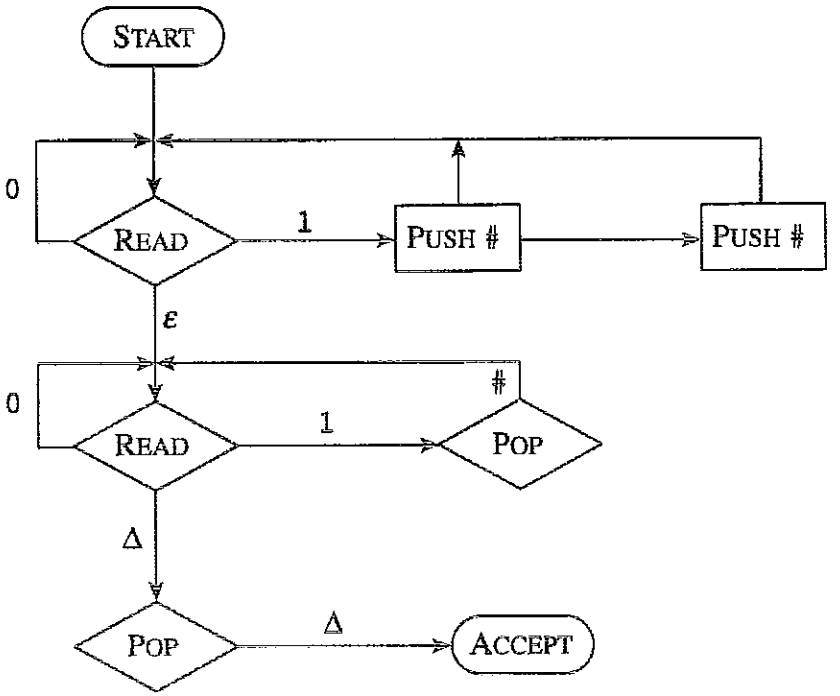
7.6 Explain how, given a PDA for L_1 and a PDA for L_2, you can produce a PDA for the concatenation $L_1 L_2$.

Ⓗ **7.7** Give a PDA for all binary strings that are *not* of the form ww.

★ **7.8** Give a PDA for the language $\{ 0^m 1^n : m = n \text{ or } m = 2n \}$.

7.9 Construct a PDA for the language of binary strings with an unequal number of 0's and 1's.

7.10 Consider the following (nondeterministic) PDA.



a) List **three** strings the PDA accepts.

b) Describe in English the language accepted by the machine.

7.11 Construct PDAs for the languages of Exercise 6.5.

★ **7.12** Construct PDAs for the languages of Exercise 6.6.

7.13 Consider the language of strings where (at least) the first half is all the same symbol. To be specific, consider the set of all strings $\mathtt{a}^n x$ where $|x| \leq n$ and $x \in \{\mathtt{a}, \mathtt{b}\}^*$. Give both a grammar and a PDA for this language.

7.14 Suppose M_1 is a PDA for language L_1 and M_2 is a PDA for language L_2. Explain how to build a PDA for the language $L_1 \cup L_2$.

7.15 Construct a PDA for the language of Exercise 6.7.

★ **7.16** Describe in English a PDA for the language of Exercise 6.8.

7.17 Describe in English a PDA accepting the language L as follows:

> $L = \{ w_1 \# w_2 \# \ldots \# w_k \# w : w, w_i \in \{0, 1\}^*$, and $w_j = w^R$ for some $j \}$. That is, for a string to be in L, it must have alphabet a subset of $\{0, 1, \#\}$, and when cut up into pieces by omitting the #, the final piece is the reverse of one of the other pieces.

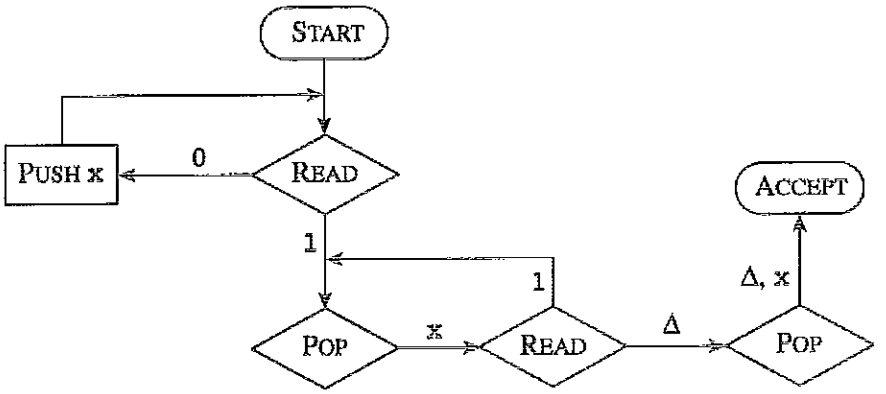
7.18 Consider the following description of a PDA. Draw the PDA as a flowchart, and explain in English what this PDA accepts:

> Until you reach the end of the input do the following: if a 1 is read, then push it; and if a 0 is read, then pop, checking that the stack does not underflow. When the end of the input is reached, accept provided the stack is not empty.

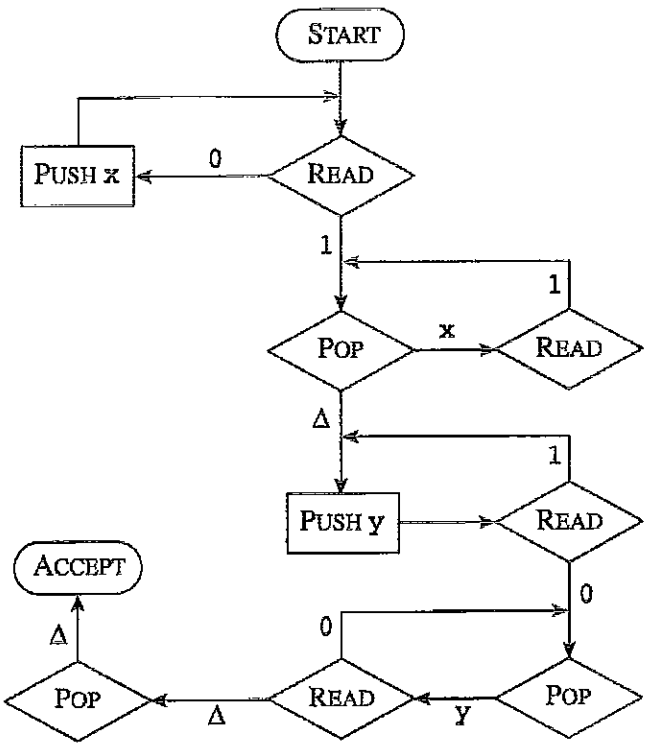
7.19 Convince your dog that a PDA can do anything an FA can do.

"For You to Do" Exercise Solutions

1.



2.



chapter

8

Grammars and Equivalences

In this chapter, we consider the Chomsky hierarchy, which includes more specific and more general grammars than context-free grammars. We also provide the proof that CFGs and PDAs have the same power.

8.1 Regular Grammars

We noted earlier that every regular language has a CFG. In fact, a regular language (without the empty string) can be generated by a grammar of a special form called a **regular grammar**. This is a grammar in which every production is of the form $A \to \mathtt{b}C$ or $A \to \mathtt{a}$ (where a and b are arbitrary terminals and C is an arbitrary variable).

Theorem Every regular language is generated by a regular grammar.

Proof

The idea is to produce a grammar such that a derivation of a string mimics the feeding of that string into the automaton. At every stage of the derivation there will be a single variable, and that variable will remember the state of the automaton.

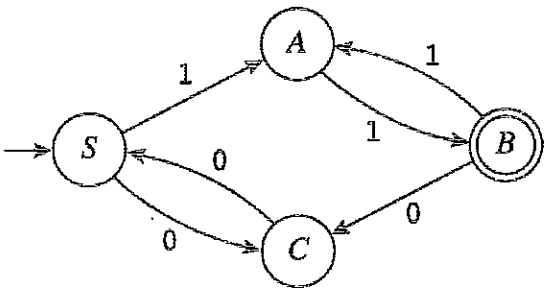
In particular, proceed as follows. Start with the DFA for the language. Introduce one variable for each state. For each transition, add a production: if $\delta(A, \mathtt{x}) = B$, then add the production $A \to \mathtt{x}B$. For each transition ending at an accept state, add a further production: if B is an accept state in the

preceding, then also add the production $A \to x$. The start variable is the start state. ♦

We illustrate the theorem with an example FA. Actually, an NFA is fine, but adding ε-transitions makes the process a bit muddier, so we do one without.

Example 8.1 Here is the construction of the regular grammar for the language of the RE $(11+00)^*11$.

Here is an NFA for the language:



This yields the following regular grammar:

$$S \to 0C \mid 1A$$
$$A \to 1B \mid 1$$
$$B \to 0C \mid 1A$$
$$C \to 0S$$

For You to Do!

1. Draw an FA, and from there write down a regular grammar for the language given by the RE 00^*11^* (first considered in Example 6.4).

8.2 The Chomsky Hierarchy

In his study of natural languages, Chomsky introduced a hierarchy of grammars. These allow for productions of various types:

0. **Unrestricted grammars.** Productions have the form $u \to v$ where u and v are any strings of terminals and/or variables.

1. **Context-sensitive grammars.** Productions have the form $xAz \rightarrow xyz$, where x, y, and z are strings of terminals and/or variables, and A is a variable. This means that A can be replaced, provided it is correctly surrounded; that is, it is in the correct context.
2. **Context-free grammars.** Productions have the form $A \rightarrow v$, where A is a variable and v a string.
3. **Regular grammars.** Productions have the form $A \rightarrow \mathtt{b}C$ or $A \rightarrow \mathtt{a}$, where a and b are terminals and A and C are variables.

We have seen that regular grammars are accepted by FAs, and that context-free grammars are accepted by PDAs. We will see later machines for the other two types of grammars.

Context-sensitive grammars are intricate affairs. Here is a famous example:

Example 8.2

A context-sensitive grammar for $0^n1^n2^n$.

$$S \rightarrow 0BS2 \mid 012$$
$$B0 \rightarrow 0B$$
$$B1 \rightarrow 11$$

See if you can provide the derivation for 000111222.

8.3 Usable and Nullable Variables

It will be useful to manipulate grammars, especially to simplify them. Two ideas that arise are usable and nullable variables. A variable is said to be **usable** if it produces some string of terminals. A variable is said to be **nullable** if the empty string is one of the strings it generates.

Example 8.3

In the following grammar, A and B are usable, but only B is nullable.

$$A \rightarrow 0A \mid 1B \mid 2C$$
$$B \rightarrow 0B \mid \varepsilon$$
$$C \rightarrow 1C$$

There is an algorithm for determining if a variable in a CFG is nullable. Indeed, the algorithm determines all nullable variables.

Identification of nullable variables	Initialize all variables as not-nullable. Repeat: Go through all productions, and if any has right-hand side empty or all entries nullable, Then mark the left-hand side variable as nullable Until there is no increase in the set of nullable variables.

A similar procedure can be used to determine the usable variables, but we leave it as an exercise.

8.4 Conversion from CFG to PDA

We turn next to proving the theorem that a language is generated by a context-free grammar if and only if it is accepted by a pushdown automaton. The proof consists of two conversions. We start with the conversion from CFG to PDA.

The basic idea in going from a CFG to a PDA is that the PDA guesses the leftmost derivation and then checks that it is correct. This is a **proof by simulation.**

For example, we gave in Example 6.6 a leftmost derivation of the string 011100. Each step of the derivation is a string. The machine guesses the next step. The problem is to do this with only a stack for storage.

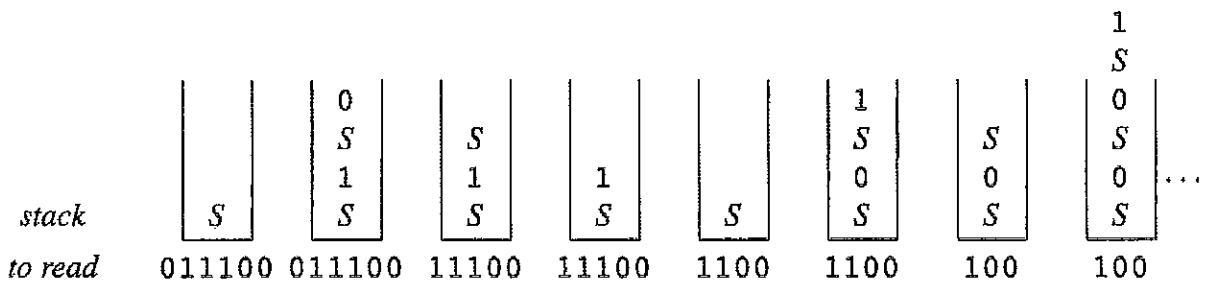
The idea is that terminals to the left of the leftmost variable do not have to be stored; instead they can be matched with the input string. The portion of the current string from the leftmost variable onward is then stored on the stack, with the *leftmost* symbol on top.

The PDA works as follows. It begins with the start variable on the stack. At each step, the PDA looks at the top of the stack:

- If there is a variable on the top, the PDA guesses a production and replaces the variable by the right-hand side of that production.
- If there is a terminal on the top, then the PDA pops it and reads one symbol from the input string and checks that the two match. (If not, this branch dies.)

When the stack is empty, the machine then checks that the input string is finished. If so, it accepts; else, it dies. Note that the machine is not guaranteed to halt.

For example, consider the CFG of Example 6.6 that generates all binary strings with an equal number of 0's and 1's. The sequence of stacks for the accepting branch of the PDA for the string 011100 starts as follows:

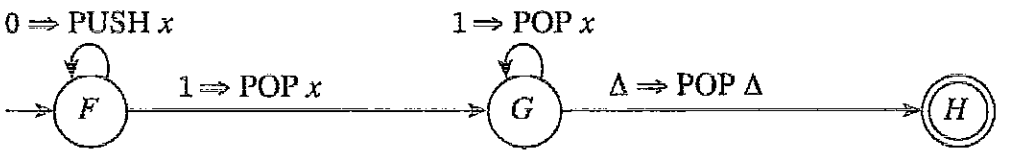


8.5 An Alternative Representation

We show next how to convert from a PDA to a CFG. This procedure is slightly easier if the PDA is in another form, where only the reading states are retained and the stack operations become part of the transitions.

Example 8.4

Consider our first PDA, given in Figure 7.1. That PDA has essentially three states: if we let F denote the first READ, let G denote the second READ, and let H denote the final accept state, then we get the following diagram for the PDA:



In this light, we can also provide a **formal definition** of a PDA. Specifically, a pushdown automaton is a 7-tuple $(Q, \Sigma, \Gamma, q_0, h_a, \Delta, \delta)$ where:

- Q is a finite set of states.
- Σ is the finite input alphabet.
- Γ is the finite stack alphabet.
- q_0 is the start state.
- h_a is the accept state.
- Δ is the empty-stack and end-of-string symbol.
- δ is the transition function.

The transition function is a function from $Q \times (\Gamma \cup \{\Delta\}) \times (\Sigma \cup \{\varepsilon, \Delta\})$ to finite subsets of $Q \times \Gamma^*$. That is, it looks at the current state, at the

current symbol on top of the stack, and possibly at the input symbol, and then proceeds to update the state and replace the current symbol on top of the stack by a string of symbols. In this formulation,

> *A pop is equivalent to replacing the stack-top symbol by nothing.*
>
> *A push is equivalent to replacing the stack-top symbol by itself and a new symbol.*

For example, $\delta(q, \mathtt{A}, \mathtt{a}) = \{(p, \mathtt{AB})\}$ means that in state q with A on top of the stack, on reading a the machine should push symbol B and change to state p.

8.6 Conversion from PDA to CFG

The idea for the conversion from PDA to CFG is to make each step in a derivation correspond to a move by the PDA. This is not trivial!

We use the alternative representation from the previous section. It helps to modify the machine further. Specifically, we assume that the stack of the PDA M is empty if and only if M is in the accept state. Further, we assume that every move is either a push of a single symbol or a pop of a single symbol (changing state while doing neither is not allowed). It is left as an exercise to show that you can massage a PDA into this form.

The variables of the CFG will be *all* the ordered triples $\boxed{q\,\mathtt{A}\,p}$ where q, p are states of the PDA, and A is a symbol from the stack alphabet. The aim is to construct the grammar such that for all strings w:

$$\boxed{q\,A\,p} \stackrel{*}{\Longrightarrow} w \quad \Leftrightarrow \quad \textit{M can go from state } q \textit{ to state } p \textit{ while reading string } w \textit{ with the net effect of popping the symbol } A \textit{ from the stack.}$$

If q_0 is the start state and h_a is the accept state of M, then $[q_0, \Delta, h_a]$ will be the start variable for the CFG: it will generate string w if and only if it is possible to go from q_0 to h_a while reading w and popping the empty-stack symbol; that is, w is accepted.

The moves of the PDA are converted to productions. There are two possible moves of the PDA, push and pop. Consider a **pop** move. Say $(p, \varepsilon) \in \delta(q, \mathtt{A}, \mathtt{a})$. That is, in state q it is possible to pop symbol A while reading a and changing to state p. Here, a is either a symbol from the input alphabet or the empty string ε. Then we simply add the production:

$$\boxed{q\,\mathtt{A}\,p} \rightarrow \mathtt{a}$$

Consider a **push** move. Say $(p, \mathtt{AB}) \in \delta(q, \mathtt{A}, \mathtt{a})$. That is, in state q with A on top of the stack it is possible to push symbol B while reading a and changing to state p. Then we add the collection of productions:

$$\boxed{q\,\mathtt{A}\,r} \rightarrow \mathtt{a}\ \boxed{p\,\mathtt{B}\,s}\ \boxed{s\,\mathtt{A}\,r} \qquad \text{for every state } r \text{ and } s$$

What this says is that there are two ways to effectively erase the symbol on top of the stack. The first is to pop it; the second is to push something, and then later erase both symbols.

The rest of the proof is to show that every derivation in this grammar corresponds to a legitimate computation of the PDA. We omit the (lengthy) details.

Example 8.5

Here is an example conversion from PDA to CFG. We include this more as evidence that the method actually works rather than seriously proposing you learn how to do this. . . .
We use the PDA for $\{0^n 1^n\}$ from Example 8.4. Here goes . . .

Start symbol: $\boxed{F\,\Delta\,H}$

Popping occurs in three places:

$$\boxed{F\,\mathtt{x}\,G} \rightarrow 1$$

$$\boxed{G\,\mathtt{x}\,G} \rightarrow 1$$

$$\boxed{G\,\Delta\,H} \rightarrow \varepsilon$$

Pushing yields 18 productions of the form

$$\boxed{F\,A\,r} \rightarrow 0\ \boxed{F\,\mathtt{x}\,s}\ \boxed{s\,A\,r} \qquad \text{for each } A \in \{\mathtt{x}, \Delta\} \text{ and } r, s \in \{F, G, H\}.$$

Now note that only eight triples ever occur on the left-hand side (including six $\boxed{F\,?\,?}$). So, we can omit those productions that produce unusable triples to yield the following list:

1. $\boxed{F\,\mathtt{x}\,F} \rightarrow 0\ \boxed{F\,\mathtt{x}\,F}\ \boxed{F\,\mathtt{x}\,F}$

2. $\boxed{F \times G} \rightarrow 0 \boxed{F \times F} \boxed{F \times G}$
3. $\boxed{F \times H} \rightarrow 0 \boxed{F \times F} \boxed{F \times H}$
4. $\boxed{F \Delta F} \rightarrow 0 \boxed{F \times F} \boxed{F \Delta F}$
5. $\boxed{F \Delta G} \rightarrow 0 \boxed{F \times F} \boxed{F \Delta G}$
6. $\boxed{F \Delta H} \rightarrow 0 \boxed{F \times F} \boxed{F \Delta H}$
7. $\boxed{F \times G} \rightarrow 0 \boxed{F \times G} \boxed{G \times G}$
8. $\boxed{F \Delta H} \rightarrow 0 \boxed{F \times G} \boxed{G \Delta H}$

But now, notice that any production with left-hand side $\boxed{F \times F}$ produces a right-hand side with $\boxed{F \times F}$ in it. And so you cannot get rid of this variable once it is introduced into a derivation. That is, all of productions 1 through 6 are of no use.

This leaves us with the CFG with start variable $\boxed{F \Delta H}$:

$\boxed{F \times G} \rightarrow 1$

$\boxed{G \times G} \rightarrow 1$

$\boxed{G \Delta H} \rightarrow \varepsilon$

$\boxed{F \times G} \rightarrow 0 \boxed{F \times G} \boxed{G \times G}$

$\boxed{F \Delta H} \rightarrow 0 \boxed{F \times G} \boxed{G \Delta H}$

Whew!

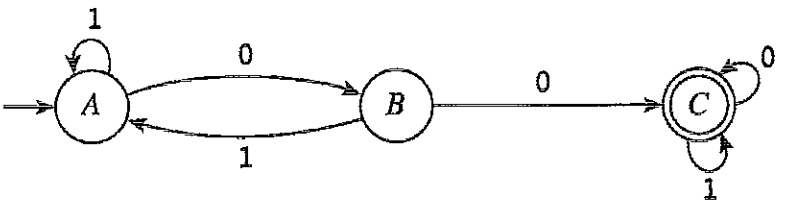
EXERCISES

8.1 Let L be the language corresponding to the RE b(ab)*. For the following CFG for L, determine the number of derivation trees for the string bababab.

$$S \rightarrow S\mathtt{a}S \mid \mathtt{b}$$

8.2 Give a regular grammar for L of the previous exercise.

8.3 Give a regular grammar for the language of the following FA (originally given in Example 1.1):



★ **8.4** Can a regular grammar be ambiguous? Explain.

8.5 Give a context-sensitive grammar for the language $\{ \mathtt{a}^i\mathtt{b}^{2i}\mathtt{a}^i \}$.

Ⓗ **8.6** Give a context-sensitive grammar for the language $\{ xx : x \in \{\mathtt{a},\mathtt{b}\}^* \}$.

8.7 Explain how to convert a PDA into the correct form for the proof of conversion to a CFG.

★ **8.8** Give a PDA equivalent to the following grammar:

$$S \to \mathtt{a}AA$$
$$A \to \mathtt{a}S \mid \mathtt{b}S \mid \mathtt{a}$$

8.9 Consider the PDA from Example 7.2 (balanced brackets).

a) In the spirit of the diagram given in Example 8.4, draw a state-based diagram for it.

b) Give the parts of this PDA in terms of the formal definition.

8.10 Suppose the stack of the PDA for language L never grows beyond 100 entries on any input. Show that L is regular.

8.11 A **2-PDA** is like a PDA except that it has two stacks.

a) Show that the language $\{ 0^n1^n2^n : n \geq 0 \}$ can be recognized by a 2-PDA.

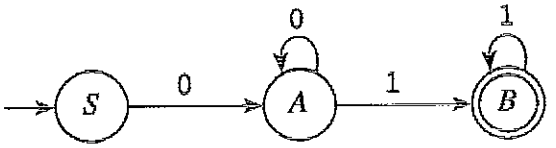
b) Show that the language $\{ 0^n1^n2^n3^n : n \geq 0 \}$ can be recognized by a 2-PDA.

c) Show that the language $\{ x\#x : x \in \{0,1\}^* \}$ can be recognized by a 2-PDA.

★ **8.12** Describe an algorithm to determine which variables in a CFG are usable. Hence, give an algorithm to determine whether the language of a CFG is empty or not.

"For You to Do" Exercise Solutions

1.



$$S \rightarrow 0A$$
$$A \rightarrow 0A \mid 1B \mid 1$$
$$B \rightarrow 1B \mid 1$$

chapter 9

Properties of Context-Free Languages

When showing the limits of regular languages, we introduced the Pumping Lemma. A similar Pumping Lemma is the tool to show a language is not context-free. In this chapter, we also consider properties of, and how to answer questions about, context-free languages.

9.1 Chomsky Normal Form

If a context-free language does not contain ϵ, then the grammar can be rewritten in Chomsky Normal Form and there is a simple (but sometimes tedious) algorithm for putting it in that form. There is also another normal form from Greibach (defined in Exercise 9.20). These normal forms can simplify matters in proving or calculating things about a grammar.

Chomsky Normal Form	A grammar where every production is either of the form $A \to BC$ or $A \to \mathtt{c}$ (where A, B, C are arbitrary variables and $\mathtt{c}$ is an arbitrary symbol).

Example 9.1 The grammar

$$S \to AS \mid \mathtt{a}$$
$$A \to SA \mid \mathtt{b}$$

is in Chomsky Normal Form.

If the language contains the empty string, then it is standard to allow $S \to \varepsilon$ where S is the start symbol, and then forbid S on the right-hand side of any production. The key advantage is that in Chomsky Normal Form, every derivation of a string of n letters has exactly $2n - 1$ steps. Thus,

> *we can determine if a string is in the language by exhaustive search of all derivations.*

Slow but sure. We will see later that this can be speeded up.

The conversion to Chomsky Normal Form has four main steps:

1. *Get rid of all ε productions.* Recall that the nullable variables are those that generate the empty string; an algorithm for finding these was given earlier in Section 8.3. Determine the nullable variables. Then, go through all productions, and for each production omit every possible subset of nullable variables. For example, if one has production $P \to AxB$ with both A and B nullable, then add the productions $P \to xB \mid Ax \mid x$. (If production P has k nullable variables on the right-hand side, then potentially $2^k - 1$ new productions are added.) After this, delete all the productions with empty right-hand sides.
2. *Get rid of all variable unit productions.* A unit production is one where the right-hand side has only one symbol. If you have the production $A \to B$, then for every production $B \to \alpha$, add the production $A \to \alpha$. Repeat until done (but don't re-create a unit production already deleted—you do have to be careful about chasing your tail).
3. *Replace every production that is too long by shorter productions.* For example, if you have the production $A \to BCD$, then replace it with $A \to BE$ and $E \to CD$. The process can be iterated with longer productions. (In theory, you introduce many new variables, but you can reuse variables if you are careful.)
4. *Move all terminals to unit productions.* For every terminal on the right of a nonunit production, add a substitute variable; for example, replace the production $A \to bC$ with the productions $A \to BC$ and $B \to b$.

Example 9.2

Consider the CFG:

$$S \to aXbX$$
$$X \to aY \mid bY \mid \varepsilon$$
$$Y \to X \mid c$$

The variable X is nullable; and so therefore is Y. After elimination of ε we obtain:

$$S \rightarrow \mathtt{a}X\mathtt{b}X \mid \mathtt{ab}X \mid \mathtt{a}X\mathtt{b} \mid \mathtt{ab}$$
$$X \rightarrow \mathtt{a}Y \mid \mathtt{b}Y \mid \mathtt{a} \mid \mathtt{b}$$
$$Y \rightarrow X \mid \mathtt{c}$$

After elimination of the unit production $Y \rightarrow X$ we obtain:

$$S \rightarrow \mathtt{a}X\mathtt{b}X \mid \mathtt{ab}X \mid \mathtt{a}X\mathtt{b} \mid \mathtt{ab}$$
$$X \rightarrow \mathtt{a}Y \mid \mathtt{b}Y \mid \mathtt{a} \mid \mathtt{b}$$
$$Y \rightarrow \mathtt{a}Y \mid \mathtt{b}Y \mid \mathtt{a} \mid \mathtt{b} \mid \mathtt{c}$$

Now, we break up the right-hand sides of S. And we replace a by A, b by B, and c by C wherever they are not units (that is, where they appear with another symbol). So we obtain the following:

$$S \rightarrow EF \mid AF \mid EB \mid AB$$
$$X \rightarrow AY \mid BY \mid \mathtt{a} \mid \mathtt{b}$$
$$Y \rightarrow AY \mid BY \mid \mathtt{a} \mid \mathtt{b} \mid \mathtt{c}$$
$$E \rightarrow AX$$
$$F \rightarrow BX$$
$$A \rightarrow \mathtt{a}$$
$$B \rightarrow \mathtt{b}$$
$$C \rightarrow \mathtt{c}$$

(Whew!)

For You to Do!

1. Convert the following CFG into Chomsky Normal Form:

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

9.2 The Pumping Lemma: Proving Languages Not Context-Free

We saw that some languages cannot be recognized by FAs. Some languages cannot be recognized even by PDAs. One example is the language of all strings of the form $0^n1^n2^n$; that is, all strings of a number of 0's followed by an equal number of 1's followed by an equal number of 2's.

But to prove this we need a Pumping Lemma. This says that a context-free language has a certain repetitiveness about it.

Pumping Lemma

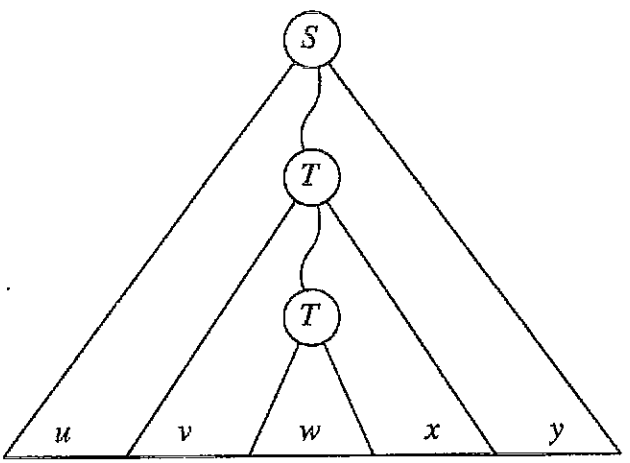
Let A be a context-free language. Then there is a constant number k such that, for every string $z \in A$ of length at least k, you can split z as $uvwxy$, where:

- vx is nonempty
- $|vwx| \leq k$
- uv^iwx^iy is in A for all $i \geq 0$

PROOF

Assume A is generated by some CFG. Consider a very long string z in the language. Then, any derivation tree for z must have many leaves; in fact, it has $|z|$ leaves. Because there is an upper bound on the number of children a node can have, this means that the tree must be very deep—there must be a long path from the root to some leaf. So, if we make z long enough, there must be a path in the tree from S to a leaf that contains the same variable twice.

Suppose that variable is T. The leaves of the subtree under the second T form a string generated by T; call this string w. The leaves of the subtree under the first T form a string containing w; let v be the substring before w and x the substring after w. Finally, the leaves of the whole tree form the whole string z, which contains vwx; let u be the substring before vwx and y the substring after.



This means the following derivations hold: that $T \stackrel{*}{\Longrightarrow} w$, that $T \stackrel{*}{\Longrightarrow} vTx$, and that $S \stackrel{*}{\Longrightarrow} uTy$.

From this it follows that $T \stackrel{*}{\Longrightarrow} v^i w x^i$ for all $i \geq 0$, and so $S \stackrel{*}{\Longrightarrow} uv^i w x^i y$ for all $i \geq 0$:

$$S \stackrel{*}{\Longrightarrow} uTy \stackrel{*}{\Longrightarrow} uvTxy \stackrel{*}{\Longrightarrow} uvvTxxy \stackrel{*}{\Longrightarrow} \cdots$$

That is, $uv^i w x^i y \in A$ for all $i \geq 0$, which is the main conclusion of the lemma.

To get a bound on the length of vwx, take T to be the lowest variable that is repeated.

The question is: how large must k be? Note that k must work for *all* input. Actually, you have to be a bit more careful. For example, if the grammar contains both $S \rightarrow T$ and $T \rightarrow S$ as productions, then a string can have an arbitrarily long derivation without working for the lemma (because we forgot to show that vx is not empty).

So, you should start the proof by normalizing the grammar, putting it into Chomsky Normal Form. In this form, every internal node in the derivation tree, except for one directly above a leaf, has two children. It follows that if the tree has more than 2^n leaves, then it has depth more than n. Thus, $k = 2^{n_A} + 1$ will work, where n_A is the number of variables in the Chomsky Normal Form grammar. We leave the details as an exercise.

We will use the notation $z^{(i)}$ to mean the string $uv^i w x^i y$.

Example 9.3

The classic example is $\{ 0^n 1^n 2^n : n \geq 0 \}$. This language is not context-free.

Let k be the constant of the Pumping Lemma. Choose the string $z = 0^k 1^k 2^k$. Consider the split of the string z into $uvwxy$. Because vwx combined has length at most k, the string vx cannot contain both 0's and 2's. This means that $z^{(0)} = uwy$ cannot have equal numbers of 0's, 1's, and 2's (why?), and is therefore not in the language. This is a contradiction.

This is the standard performance. You suppose the language is context-free. Then it must have the properties described by the Pumping Lemma. But you show that it does not have these properties for at least one z. Therefore, it is not context-free.

Example 9.4

Consider the language $\{ x\#x : x \in \{0, 1\}^* \}$. (The hash mark is a special symbol that occurs in the middle of the input string.) This language is not context-free.

Let k be the constant of the Pumping Lemma. Choose the string $z = 0^k 1^k \# 0^k 1^k$. Consider the split of the string z into $uvwxy$. Because

$z^{(0)} = uwy$ is in the language, it must still have a # in the middle; so v occurs before the middle of z and x occurs after the middle. But because vwx combined has length at most k, this means that v is a string of 1's and x is a string of 0's, and so $z^{(0)}$ is not in the language after all. This is a contradiction.

Example 9.5

But note that we do not run into a contradiction with $\{ 0^n 1^n : n \geq 0 \}$. For, if we take $z = 0^k 1^k$, then we may write $z = uvwxy$, where the string v is the last 0, the string x is the first 1, and w is empty. With such a choice, $z^{(i)} = uv^i wx^i y$ is in the language.

For You to Do!

2. Show that the language $\{ \mathtt{a}^i \mathtt{b}^j \mathtt{c}^i \mathtt{d}^j : i, j > 0 \}$ is not context-free.

EXERCISES

9.1 Convert the grammar of Example 6.9 into Chomsky Normal Form (assuming "*number*" is a terminal).

9.2 Convert the following grammar with start variable S into Chomsky Normal Form:

$$S \to AS\,\mathtt{a} \mid \mathtt{a}B$$
$$A \to B \mid S$$
$$B \to \mathtt{b} \mid \varepsilon$$

9.3 Let A be an alphabet and consider any function f that maps symbols in A to nonempty strings in A. Then, for any string w, the string $f(w)$ is defined by replacing each symbol by its f-value. For example, if $A = \{0, 1\}$ and $f(0) = 00$ and $f(1) = 11$, then $f(001) = 000011$. Then, for any language L, the language L^f is defined by $L^f = \{f(w) : w \in L\}$. (The function f is sometimes called a homomorphism.)

a) Show that if L is context-free, then so is L^f.

b) Show by example that L^f can be context-free even if L is not.

★ **9.4** Show that the language $\{\mathtt{a}^n \mathtt{b}^{2n} \mathtt{a}^n\}$ is not context-free.

9.5 Show that the language $\{ 0^a 1^b 2^c \text{ with } a < b < c \}$ is not context-free.

9.6 Show that the language $\{ xx : x \in \{0,1\}^* \}$ is not context-free.

Ⓗ **9.7** Show that the complement of the language in the previous question is context-free.

★ **9.8** Show that the complement of $\{0^n 1^n 2^n\}$ is context-free.

9.9 What do the previous two exercises tell us about closure of the set of context-free languages under complements?

9.10 Consider the set of all strings with the unary alphabet $\{\#\}$ with length a perfect square. Show that this language is not context-free.

9.11 Consider the set of all strings with the unary alphabet $\{\#\}$ with length a prime number. Show that this language is not context-free.

★ **9.12** Show by example that context-free languages are *not* closed under intersection. Hint: Start with the context-free language $\{ 0^n 1^n 2^m \}$.

9.13 Show that the intersection of a regular language and a context-free language is context-free.

Ⓗ **9.14** Show that a context-free language over a unary alphabet is regular.

9.15 **a)** Give an example of a non-context-free language L such that L^* is not context-free.

b) Give an example of a non-context-free language M such that M^* is context-free.

★ **9.16** Consider the set of all strings of the form x/y, where x and y are positive decimal integers such that $x < y$. The set of such strings represents the set of all possible decimal fractions. Determine if this is a context-free language.

9.17 A CFG is called **linear** if the right-hand side of every production contains at most one variable. Thus, a regular grammar is always linear. But a linear grammar need not generate a regular language; for example, we saw that palindromes are generated by a linear grammar.

Show that the set of languages generated by linear grammars is closed under union.

Ⓗ **9.18** (Continuation of previous question.) Prove the following Pumping Lemma for linear grammars. If language A has a linear grammar, then there is a constant number k such that for every string $z \in A$ of length at least k, you can split z as $uvwxy$, where: vx is nonempty, $|uvxy| \leq k$, and for all i, $uv^i wx^i y \in A$.

9.19 (Continuation of previous question.) Consider the language $\{ 0^i 1^i 0^j 1^j : i, j > 0 \}$.

a) Show that this language is context-free.

b) Show that this language does not have a linear grammar.

★ **9.20** A context-free grammar is in **Greibach Normal Form** if every production has a right-hand side that starts with a terminal and the rest is variables. That is, productions have the form $S \to \mathtt{a}B^*$.

Convert the following grammar into Greibach Normal Form:

$$S \to C\,S\,\mathtt{b} \mid \mathtt{aa}$$
$$C \to \mathtt{ab}\,S$$

9.21 A **counter automaton** is an FA that has access to a counter. Equivalently, it is a PDA that can only push one particular symbol onto the stack.

a) Show that the languages accepted by a deterministic counter automaton are closed under complementation.

b) Show that the languages accepted by a nondeterministic counter automaton are closed under union.

"For You to Do" Exercise Solutions

1. After the first step, you have the following:

$$S \to A\mathtt{b}A \mid \mathtt{b}A \mid A\mathtt{b} \mid \mathtt{b}$$
$$A \to A\mathtt{a} \mid \mathtt{a}$$

The second step does not apply. After the third step, you have the following:

$$S \to TA \mid \mathtt{b}A \mid A\mathtt{b} \mid \mathtt{b}$$
$$A \to A\mathtt{a} \mid \mathtt{a}$$
$$T \to A\mathtt{b}$$

And finally, you have the following:

$$S \to TA \mid BA \mid AB \mid \mathtt{b}$$
$$A \to AC \mid \mathtt{a}$$
$$T \to AB$$
$$B \to \mathtt{b}$$
$$C \to \mathtt{a}$$

2. Suppose the language is context-free. Let k be the constant of the Pumping Lemma. Choose the string $z = \mathtt{a}^k\mathtt{b}^k\mathtt{c}^k\mathtt{d}^k$. Consider the split of the string z into $uvwxy$. Because vx is nonempty, it contains some symbols. However, because vwx combined has length at most k, the string vx cannot contain both a's and c's, nor can it contain both b's and d's. Thus, $z^{(0)}$ is not in the language. This is a contradiction.

Chapter

10

Deterministic Parsing

The whole idea of grammars and derivation trees is pointless unless there is an efficient procedure for a given grammar to take the string and find a derivation. **Parsing** means getting from the string to the derivation tree and rejecting those strings that are not generated by the grammar. In this chapter, we describe a common algorithm for parsing in a compiler. We also describe a theoretical result that there is a reasonable parsing algorithm for any context-free language.

10.1 Compilers

A compiler for a language such as Java or C starts by building a derivation tree for the input program. The derivation tree reveals the structure of the input. This phase is the task of the **parser**: it essentially determines if and how the input can be derived from the start variable within the rules of the grammar.

There are two competing approaches to parsing. The easiest approach to parsing is top-down parsing. With this, you look at the first symbol of the input to determine which production was used first. If you are lucky, then only one production starts with that symbol, but in general that is not the case. Our conversion from grammar to PDA took a top-down approach, but it had nondeterminism to help it choose. In this book, however, we examine only bottom-up parsing. In this, you start with the input string and try to work backward to the start variable S.

Note that a context-free grammar is not quite sufficient to handle the syntax of a full programming language. The parts that are not describable

by such a mechanism are usually easily handled. Such context dependence occurs in rules of type checking or of repeats. For example, many languages have the rule that an identifier cannot be declared twice in the same block. This prohibition cannot be enforced by a context-free grammar. A common strategy is to construct a CFG that is slightly more permissive, accepting a larger language, and then to filter out illicit constructs. On the other hand, accurately describing a language's semantics (meaning) is more difficult.

10.2 Bottom-Up Parsing

Bottom-up parsing is also called **shift–reduce** parsing. Here you read symbols until you have a group that fully matches the right-hand side of a production. Then, you replace that with the left-hand side of the production—called a reduction. And then you continue. This produces a rightmost derivation.

The main question is: should you reduce now, or read the next symbol? We will see that if the CFG is in a certain form, you can tell whether to perform a reduction or to wait until you have more symbols and then perform a different reduction.

The parsing process consists of consecutive steps. There are two kinds of steps:

- **Shift** the next input terminal onto the stack.
- **Reduce** a stacked sequence of symbols to a single variable according to a production. That is, if the subsequence of symbols on top of the stack is the right-hand side of a production, then replace it by the corresponding variable.

The process terminates when you have only the start variable left.

Example 10.1 Recall the grammar for arithmetic expressions (see Example 6.9). Slightly modified, it is as follows:

$$E \to E + T \mid T$$
$$T \to T \times F \mid F$$
$$F \to (E) \mid \mathtt{n}$$

The parsing of $(\mathtt{n} + \mathtt{n}) \times \mathtt{n}$ is given. Note that this parsing *assumes* that the parser knows what to do at each step.

Type	*Stack*	*To be read*	*Comment*
		(n+n)*n	
s	(	n+n)*n	
s	(n	+n)*n	
R	(F	+n)*n	
R	(T	+n)*n	
R	(E	+n)*n	
s	(E+	n)*n	
s	(E+n	)*n	
R	(E+F	)*n	
R	(E+T	)*n	
R	(E	)*n	not reduction T to E
s	(E)	*n	
R	F	*n	
R	T	*n	
s	T*	n	not reduction T to E
s	T*n		
R	T*F		
R	T		not reduction F to T
R	E		

The concrete problem you face in bottom-up parsing is in determining which kind of step to take next and, in the case of a reduce step, which production to use. This question is not easily answered.

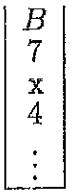
It is clear that a nondeterministic PDA can implement a bottom-up parser for any grammar: at each stage it makes the correct choice between shift and reduce, and if reduce, then the correct choice of production. Unfortunately, you cannot always get rid of the nondeterminism. And we most certainly have only deterministic programs. Knuth determined which languages can be recognized by deterministic PDAs. They are given by what are called LR grammars. So, we have to restrict the types of grammars.

10.3 Table-Driven Parser for LR(1) Grammars

An **LR(1) grammar** is defined as a CFG where looking one symbol ahead is enough to decide whether to shift or to reduce.

For efficient parsing, the information on which the decision is to be made must be presented in an appropriate fashion. In the LR(1) parser, this information is given in a **table**. The table tells you if the next operation should be a reduce or a shift. It also has information about the next state of the parser.

In particular, the parser PDA has a current state and a stack. Apart from having terminals on it, the stack is also used to keep track of the state: the entries on the stack alternate between states and symbols (terminal or variable). For example, the stack might be as follows:



We will number the states.

For each state and each possible next input symbol, the table specifies one of four possible actions:

- *Shift.* (1) Push the current state onto the stack. (2) Read the next symbol and push it onto the stack. (3) Change to the state specified in the table.
  For example: the table entry "s4" means shift and change to state 4.
- *Reduce by the specified production.* (1) Pop the symbols and the intervening states off the stack and discard. (2) Update the state as follows. Say you have reduced to variable X. Look at, but do not pop, the state on top of the stack: say it is q. Then look up in the table what the new state is based on q and X. (3) Push X.
  For example: the entry "R1" means use production 1. Say this is $C \rightarrow$ xB and the stack is as in the preceding picture. Then the reduction pops the B, the state symbol 7 and the x, and discards the three. Now it notes the stack top is 4, pushes the C, and updates its state to that given in row 4 column C.
- *Error.* A blank entry in the table indicates an error in the input.
- *Halt.* And accept.

Example 10.2 Consider the simplified grammar for variable declarations (from Hunter):

1. $S \rightarrow$ `real` $IDLIST$
2. $IDLIST \rightarrow IDLIST$, ID
3. $IDLIST \rightarrow ID$
4. $ID \rightarrow$ `A` | `B` | `C` | `D`

To make it easier to follow, we abbreviate. (In a real parser, the input will already have been converted into such tokens.)

1. $S \rightarrow \mathtt{r}L$
2. $L \rightarrow L\,,I$
3. $L \rightarrow I$
4. $I \rightarrow \mathtt{v}$

Each state of the parser corresponds to partial right-hand sides of some productions. This is discussed in more detail later; in this table, we summarize this as "progress". Table 10.1 is the LR(1) table (where eos stands for end-of-string).

State	*Progress*	r	,	v	eos	*S*	*I*	*L*
0		s1				2		
1	r			s3			4	5
2	*S*				*acc*			
3	v		R4		R4			
4	*I*		R3		R3			
5	r*L*		s6		R1			
6	*L* ,			s3			7	
7	*L* ,*I*		R2		R2			

Table 10.1

On input `real A,B`—which we abbreviate `rv,v`—the parser proceeds as follows:

Curr state	*Stack*	*To be read*	*Operation to apply*
0		rv,v	s1
1	0 r	v,v	s3
3	0 r 1 v	,v	R4
4	0 r 1 *I*	,v	R3
5	0 r 1 *L*	,v	s6
6	0 r 1 *L* 5 ,	v	s3
3	0 r 1 *L* 5 , 6 v		R4
7	0 r 1 *L* 5 , 6 *I*		R2
5	0 r 1 *L*		R1
2	0 *S*		acc

Example 10.3 Recall the grammar from Example 10.1. Let us number the productions:

1. $E \to E + T$
2. $E \to T$
3. $T \to T \times F$
4. $T \to F$
5. $F \to (E)$
6. $F \to \mathtt{n}$

The table (based on Parsons):

State	*Progress*	+	×	(	)	n	eos	*E*	*T*	*F*
0				s1		s2		3	4	5
1	(			s1		s2		6	4	5
2	n	R6	R6		R6		R6			
3	E	s7					*acc*			
4	T	R2	s8		R2		R2			
5	F	R4	R4		R4		R4			
6	$(E$	s7			s9					
7	$E +$			s1		s2			10	5
8	$T \times$			s1		s2				11
9	(E)	R5	R5		R5		R5			
10	$E + T$	R1	s8		R1		R1			
11	$T \times F$	R3	R3		R3		R3			

Table 10.3

You should at least check that some of the reductions and shifts make sense. For example, if in state 7 an n is read, then it is shifted (so that n is pushed onto the stack) and the progress becomes just the n.

10.4 Construction of an SLR(1) Table

We now describe how to build the table. Technically, this process assumes the grammar is an SLR(1) grammar (S for simple). This process has two main phases: (1) determining the states and the shifts, and (2) determining the reductions. But we start by explaining what each state corresponds to. We use the arithmetic grammar as an example.

10.4.1 Defining the States

The key construct is called an **LR(0)-item**, but we will just call it an **item**. Each item is a production with a certain point on the right-hand side marked—a dot acts as the **marker** and the whole item is enclosed in square brackets. For example $[E \to E + \cdot T]$ is the production $E \to E + T$ with a marker before the T.

If a production has a right-hand side of length n, then there are $n + 1$ items corresponding to that production. For example, if $S \to \varepsilon$, then there is one item, which is written $[S \to \cdot]$. An **initial** item is one where the marker is at the left end; a **completed** item is one where the marker is at the right end.

The intended meaning of an item is that the symbols to the left of the marker have already been read, or reduced to, and they are the top of the stack. It turns out that these items can form the states of a nondeterministic machine. But we need the deterministic version, so we shall go straight there. For our parser,

each state is a set of items, called an ***item-set.***

It is to be noted that, if the construction goes wrong at any point, that is proof that the grammar is not in the right form. In this book, we deal only with grammars that are of the right form.

The main operation is to determine the item that comes after a given item. This corresponds to a shift from earlier.

1. *Advance the marker.* So, if the item is $[E \to \cdot (E)]$ and the next symbol is (, go to the item $[E \to (\cdot E)]$.
2. *Take the closure.* For an item, the **closure** is the set of items as follows: if the marker precedes a variable, then the closure adds all initial items starting with that variable. If one of the new items has a marker preceding a variable, then add all initial items with that variable. And repeat. If the marker precedes a nonterminal, there is nothing to add.

Example 10.4 For the arithmetic grammar, consider the item $[T \to T \cdot \times F]$. On the symbol $\times$, the resulting item is $[T \to T \times \cdot F]$. The closure adds all initial items beginning with F: that is, $[F \to \cdot (E)]$ and $[F \to \cdot \mathtt{n}]$.

In general, to find the transition from an item-set rather than a single item, do this process with each item and take the union of the resultant closures.

Determining the next item-set

Let Q be an item-set and σ a symbol. The next item-set is constructed by the following:

For each item in Q where the marker precedes σ:

1. Add the item that results from advancing the marker past σ.
2. Add the closure of the resultant item.

For the parser to recognize when it is finished, it is standard practice to add a new start variable with its only production taking it to the old start variable. If ever you reduce to the new start variable, you are done. So, in the example, add production 0, which is $E' \rightarrow E$. The start item-set is the initial item for the start variable and the closure of this item.

Example 10.5

The start item is $[E' \rightarrow \cdot E]$. Its closure adds all initial items starting with E, which in turns adds all initial items starting with T, which in turn adds all initial items starting with F. The start item-set is therefore $\{[E' \rightarrow \cdot E], [E \rightarrow \cdot E + T], [E \rightarrow \cdot T], [T \rightarrow \cdot T \times F], [T \rightarrow \cdot F], [F \rightarrow \cdot (E)], [F \rightarrow \cdot \mathtt{n}]\}$.

10.4.2 Determining the States and Shifts in the Table

We now determine the states and the shifts. Start with the start item-set. Consider every possible next symbol: terminal and variable. For each symbol, determine the next item-set. Then, for each item-set so created, do the same thing. (This process should remind you of the conversion from an NFA with ε-transitions to a DFA.) Repeat until no new item-set is created.

State number	When first occurs (state, symbol)	Items found by moving marker	Initial items added by closure
0	start	$[E' \rightarrow \cdot E]$	E, T, F
1	0, (	$[F \rightarrow (\cdot E)]$	E, T, F
2	0, n	$[F \rightarrow \mathtt{n} \cdot]$	
3	0, E	$[E' \rightarrow E \cdot]$, $[E \rightarrow E \cdot + T]$	
4	0, T	$[E \rightarrow T \cdot]$, $[T \rightarrow T \cdot \times F]$	
5	0, F	$[T \rightarrow F \cdot]$	
6	1, E	$[F \rightarrow (E \cdot)]$	
7	3, +	$[E \rightarrow E + \cdot T]$	T, F
8	4, ×	$[T \rightarrow T \times \cdot F]$	F
9	6,)	$[F \rightarrow (E) \cdot]$	
10	7, T	$[E \rightarrow E + T \cdot]$, $[T \rightarrow T \cdot \times F]$	
11	8, F	$[T \rightarrow T \times F \cdot]$	

(The second column is added just to help understand the algorithm.)

At the same time, you start on the table (Table 10.2) from earlier. Each of these transitions produces a shift given in the earlier table. For example, suppose in state 3 you consider the transition for +. This defines a new item-set that produces state 7. A shift s7 is added to the table. Note too that some states recur; for example, for state 7 on symbol n, you go back to state 2.

10.4.3 Determining the Reductions

Armed with the states and the shifts, the second phase is to determine the reductions. At one level, the reductions are obvious. Take each completed item and make that a reduction. The issue is that the fundamental question you are trying to answer is *when to reduce and when not to.* And how to recognize an error in the input.

We need the concept of FOLLOW sets. For a variable A, FOLLOW(A) is a set of terminals; specifically, it is the set of all possible terminal symbols that can follow A in a string derived from S. (To be strictly accurate, this definition is only correct if every variable yields at least one string, but that's always the case here.) You include the eos symbol Δ in FOLLOW(A) if the variable A can occur at the end of a derived string. For example, if $S \stackrel{*}{\Longrightarrow} BC\mathtt{x}BA\mathtt{y}\mathtt{x}C$, then you add y to FOLLOW(A) and Δ, x to FOLLOW(C).

Example 10.6 In the arithmetic grammar, FOLLOW(E) is $\{), \Delta, +\}$. Both FOLLOW(T) and FOLLOW(F) are $\{), \Delta, +, \times\}$. You should verify this. For instance, $E' \stackrel{*}{\Longrightarrow} E \stackrel{*}{\Longrightarrow} T \stackrel{*}{\Longrightarrow} T \times F \stackrel{*}{\Longrightarrow} F \times F$ shows that $\times \in$ FOLLOW(F).

You can now add the reductions. Specifically:

> *For each completed item, say an A-item, add a reduction to the table for all symbols that occur in* FOLLOW(A). *Except that the reduction corresponding to the artificial production $E' \rightarrow E$ becomes the acceptance point.*

You can then check that the reductions in Table 10.2 are correct. Note that in no case did a reduction occur where there was already a shift.

This process sounds rather daunting. Fortunately, there is software to do it. The UNIX utility YACC generates LR tables for certain CFGs. For more information, consult the ultimate book by Aho, Sethi, and Ullman, or play around with JFLAP (see page 50).

For You to Do!

1. For the grammar of Example 10.2:
   a) Verify that FOLLOW$(S) = \{\Delta\}$ and FOLLOW(L) = FOLLOW$(I) = \{,, \Delta\}$.
   b) Determine the item-sets.
   c) Verify the entries in Table 10.1.

10.5 Guaranteed Parsing

In this section, the goal is an algorithm for any context-free language that determines whether an input string is in the language. This result is mostly of theoretical interest, because the resultant parser is too slow for most practical situations.

The algorithm is called the CYK algorithm, named after Cocke, Younger, and Kasami. It uses a technique known as **dynamic programming**. Dynamic programming is similar to recursion, but it works bottom-up.

The algorithm assumes the context-free language is given by a CFG G in Chomsky Normal Form. If not, the preliminary preprocessing step is to convert the grammar to this form.

Consider an input string w of length n that you want to test whether G generates it. The key idea is to solve a more general problem: you look at substrings of w. Say $w = w_1 w_2 \dots w_n$. Then, for $1 \leq i \leq j \leq n$, let $w_{i,j}$ be the substring $w_i w_{i+1} \dots w_j$. The problem you solve is as follows:

which variables produce which substrings.

The point is that suppose $n \geq 2$ and the start variable S produces w. Because G is in Chomsky Normal Form, the first step in the derivation of w must be a production that replaces S by two variables: say $S \to AB$. Then, it follows that the string w can be split into two pieces: the first piece generated from A and the second generated from B. So, the algorithm to determine whether $S \stackrel{*}{\Longrightarrow} w$ is to determine whether there is a split k and a production $S \to AB$ such that $A \stackrel{*}{\Longrightarrow} w_{1,k} = w_1 w_2 \dots w_k$ and $B \stackrel{*}{\Longrightarrow} w_{k+1,n} = w_{k+1} w_{k+2} \dots w_n$.

But how to determine whether $A \stackrel{*}{\Longrightarrow} w_{1,k}$? Thinking recursively, we see that the same idea continues. In general, suppose you want to know if variable $A \stackrel{*}{\Longrightarrow} w_{i,j}$ for some i and j. Consider all possible productions from A. For each such production, say $A \to EF$, try all possible k from i up to $j - 1$. For each k, ask whether $E \stackrel{*}{\Longrightarrow} w_{i,k}$ and $F \stackrel{*}{\Longrightarrow} w_{k+1,j}$.

To make this efficient, you work from the bottom up; that is, answer the question for smaller strings first. A **table** is used to keep track of the answers so far.

CYK algorithm

1. Start by answering for each i and each variable A whether $A \stackrel{*}{\Longrightarrow} w_{i,i}$. In this case, the substring $w_{i,i}$ is just a single symbol; so you simply look at the unit productions of G.
2. Then answer for each i and each variable A whether $A \stackrel{*}{\Longrightarrow} w_{i,i+1}$. In this case, you use the general recipe provided earlier.
3. Repeat for all $w_{i,i+2}$, then for all $w_{i,i+3}$, and so on. At each stage, determine completely the set of variables that produces the substrings of that length, and then increase the length.

At the final stage, you determine the variables for substrings of length n. Well, actually $w = w_{1,n}$ itself is the only such substring. This enables you to answer the original question: whether $S \stackrel{*}{\Longrightarrow} w$ or not.

Example 10.7

Consider the CFG with start variable S:

$$S \to ST \mid TU \mid \mathtt{b}$$
$$T \to SU \mid \mathtt{a}$$
$$U \to SS \mid \mathtt{b}$$

Consider the input string $w = \mathtt{aababb}$.

The table constructed is as follows:

start \ finish	1	2	3	4	5	6
1	T	.	.	.	S	S,T,U
2		T	S	S	S,T,U	S,T,U
3			S,U	S	T,U	S,T,U
4				T	S	S,T,U
5					S,U	T,U
6						S,U

What does this table mean? Consider for example the entry in row 3 column 5. This entry says that variables T and U generate the string $w_{3,5}$ (which is bab). This entry was calculated by the algorithm. For example, the variable T is here because $T \to SU$ (one of the productions) and $S \stackrel{*}{\Longrightarrow} w_{3,4}$, $U \stackrel{*}{\Longrightarrow} w_{5,5}$ (look to the left of and down from this entry).

Because S is in the entry in row 1 column 6, it follows that w is in the language. With a bit more effort you can read off a derivation from

the table. This entails recording (or recalculating) where the entries came from. For example, the S is there since $T \stackrel{*}{\Longrightarrow} w_{1,1}$ and $U \stackrel{*}{\Longrightarrow} w_{2,6}$. The U in the entry for $w_{2,6}$ is there for two splits, including $S \stackrel{*}{\Longrightarrow} w_{2,3}$ and $S \stackrel{*}{\Longrightarrow} w_{4,6}$. And so on.

For You to Do!

2. For the abbreviated grammar of Example 10.2, convert to Chomsky Normal Form, and then apply the CYK algorithm to the string `rv,v,v`.

EXERCISES

10.1 Consider the CFG

1. $S \rightarrow \mathtt{a}\mathtt{a}S$
2. $S \rightarrow \mathtt{b}$

The SLR(1) table is

State	a	b	eos	S
0	s1	s2		3
1	s4			
2			R2	
3			*acc*	
4	s1	s2		5
5			R1	

a) Show the steps in parsing aaaab.
b) Show the steps in parsing aaabab (which is not in the language).
c) Determine the item-sets.
d) Determine FOLLOW(S).

10.2 Consider the CFG with start variable S:

1. $S \rightarrow AX$
2. $A \rightarrow \mathtt{a}$
3. $A \rightarrow \mathtt{b}$
4. $X \rightarrow \mathtt{x}$
5. $X \rightarrow \mathtt{y}$

Derive the item-sets and the table for an LR-parser for this grammar.