ÜLEYMAN DEMIREL UNIVERSITY
Faculty of Engineering
Department of Computer Science

Fundamental Structures OF Computer Science

(Course materials)

Part 1

Introduction to Automata and Formal Languages

(*Finite Memory Programs, Finite Automata, Recursion, and Formal Languages and Grammars*)

Almaty ◦ 2001

SÜLEYMAN DEMIREL UNIVERSITY
Faculty of Engineering
Department of Computer Science

Fundamental Structures OF Computer Science

(Course materials)

Part 1

Introduction to Automata and Formal Languages

(*Finite Memory Programs, Finite Automata, Recursion, and Formal Languages and Grammars*)

Almaty • 2001

ББК 32.973-01я73
П27

Mikhail G.Peretyat'kin
П27 "Fundamental Structures of Computer Science (Course materials)
Part 1. Introduction to Automata and Formal Languages",
Almaty – 2002, 166 pages
ISBN 9965-9150-4-0

ББК 32.973-01я73

П $\frac{4310020000}{00(05)-02}$

ISBN 9965-9150-4-0

Serial Parts of the Course:

Part 1 :
Introduction to Automata and Formal Languages [1], [5]
(Finite Memory Programs, Finite Automata, Recursion, and Formal Languages and Grammars)

Part 2 :
Data Structures and Algorithm Analysis [2], [3], [4] [5]
(analysis, correctness, lists, stacks, queues, trees, hashing, heaps, sorting)

References

1. EITAN GURARI, ***An introduction to the Theory of Computation***, Computer Science Press, 1989. (Lecture notes, Ohio University, Spring 1999, 321p.).
2. JEFFRY H. KINGSTON, ***Algorithms and Data Structures (Design, Correctness, Analysis)***. Addison-Wesley, Second edition, 1998, 380p.
3. MARK ALLEN WEISS, ***Data Structures and Algorithm Analysis in C***, Addison-Wesley, Second edition, 511p.
4. ROBERT L. KRUSE, ***Data Structures and Program Design***, Prentice Hall of India, New Delhi–110 001, 1996, Third Edition, 689p.
5. KENNETH H. ROSEN, ***Discrete Mathematics and its Applications***, McGraw-Hill, Mathematics series, 1995, 709p.

The course materials are prepared and assembled by
Mikhail G. Peretyat'kin

PREFACE

Computations are designed to solve problems. Programs are descriptions of computations written for execution on computers. The field of computer science is concerned with the development of methodologies for designing programs, and with the development of computers for executing programs. It is therefore of central importance for those involved in the field that the characteristics of programs, computers, problems, and computation be fully understood. Moreover, to clearly and accurately communicate intuitive thoughts about these subjects, a precise and well-defined terminology is required.

This book explores some of the more important terminologies and questions concerning programs, computers, problems, and computation. The exploration reduces in many cases to a study of mathematical theories, such as those of automata and formal languages; theories that are interesting also in their own right. These theories provide abstract models that are easier to explore, because their formalisms avoid irrelevant details.

The material in this book gradually increases in complexity. In many cases, new topics are treated as refinements of old ones, and their study is motivated through their association to programs.

Chapter 1 is concerned with the definition of some basic concepts. It starts by considering the notion of strings, and the role that strings have in presenting information. Then it relates the concept of languages to the notion of strings, and introduces grammars for characterizing languages. The chapter continues by introducing a class of programs. The choice is made for a class, which on one hand is general enough to model all programs, and on the other hand is primitive enough to simplify the specific investigation of programs. In particular, the notion of nondeterminism is introduced through programs. The chapter concludes by considering the notion of problems, the relationship between problems and programs, and some other related notions.

Chapter 2 studies finite-memory programs. The notion of a state is introduced as an abstraction for a location in a finite-memory program as well as an assignment to the variables of the program. The notion of state is used to show how finite-memory programs can be modeled by abstract computing machines, called finite-state transducers. The transducers are essentially sets of states with rules for transition between the states. The inputs that can be recognized by finite-memory programs are characterized in terms of a class of grammars, called regular grammars. The limitations of finite-memory programs, closure properties for simplifying the job of writing finite-memory programs, and decidable properties of such programs are also studied.

Chapter 3 considers the introduction of recursion to finite-memory programs.

The treatment of the new programs, called recursive finite-domain programs, resembles that for finite-memory programs in Chapter 2. Specifically, the recursive finite-domain programs are modeled by abstract computing machines, called pushdown transducers. Each pushdown transducer is essentially a finite-state transducer that can access an auxiliary memory that behaves like a pushdown storage of unlimited size. The inputs that can be recognized by recursive finite-domain programs are characterized in terms of a generalization of regular grammars, called context-free grammars. Finally, limitations, closure properties, and decidable properties of recursive finite-domain programs are derived using techniques similar to those for finite-memory programs.

Chapter 4 deals with the general class of programs. Abstract computing machines, called Turing transducers, are introduced as generalizations of pushdown transducers that place no restriction on the auxiliary memory. The Turing transducers are proposed for characterizing the programs in general, and computability in particular. It is shown that a function is computable by a Turing transducer if and only if it is computable by a deterministic Turing transducer. In addition, it is shown that there exists a universal Turing transducer that can simulate any given deterministic Turing transducer. The limitations of Turing transducers are studied, and they are used to demonstrate some undecidable problems. A grammatical characterization for the inputs that Turing transducers recognize is also offered.

The choice of topics for the text and their organization are generally in line with what is the standard in the field. The exposition, however, is not always standard. For instance, transition diagrams are offered as representations of pushdown transducers and Turing transducers. These representations enable a significant simplification in the design and analysis of such abstract machines, and consequently provide the opportunity to illustrate many more ideas using meaningful examples and exercises. The level of the material is intended to provide the reader with introductory tools for understanding and using formal specifications in computer science. As a result, in many cases ideas are stressed more than detailed argumentation, with the objective of developing the reader's intuition toward the subject as much as possible.

Theorems, Figures, Exercises, and other items in the text are labeled with triple numbers. An item that is labeled with a triple `i.j.k` is assumed to be the kth item of its type in Section j of Chapter i.

Chapter 1

GENERAL CONCEPTS

Computations are designed for processing information. They can be as simple as an estimation for driving time between cities, and as complex as a weather prediction.

The study of computation aims at providing an insight into the characteristics of computations. Such an insight can be used for predicting the complexity of desired computations, for choosing the approaches they should take, and for developing tools that facilitate their design.

The study of computation reveals that there are problems that cannot be solved. And of the problems that can be solved, there are some that require infeasible amount of resources (e.g., millions of years of computation time). These revelations might seem discouraging, but they have the benefit of warning against trying to solve such problems. Approaches for identifying such problems are also provided by the study of computation.

On an encouraging note, the study of computation provides tools for identifying problems that can feasibly be solved, as well as tools for designing such solutions. In addition, the study develops precise and well-defined terminology for communicating intuitive thoughts about computations.

The study of computation is conducted in this book through the medium of programs. Such an approach can be adopted because programs are descriptions of computations.

Any formal discussion about computation and programs requires a clear understanding of these notions, as well as of related notions. The purpose of this chapter is to define some of the basic concepts used in this book. The first section of this chapter considers the notion of strings, and the role that strings have in representing information. The second section relates the concept of languages to the notion of strings, and introduces grammars for characterizing languages. The third section deals with the notion of programs, and the concept of nondeterminism in programs. The fourth section formalizes the notion of problems, and discusses the relationship between problems and programs. The fifth section defines the notion of reducibility among problems.

1.1 Alphabets, Strings, and Representations

The ability to represent information is crucial to communicating and processing information. Human societies created spoken languages to communicate on a basic level, and developed writing to reach a more sophisticated level.

The English language, for instance, in its spoken form relies on some finite set of basic sounds as a set of primitives. The words are defined in term of finite sequences of such sounds. Sentences are derived from finite sequences of words. Conversations are achieved from finite sequences of sentences, and so forth.

Written English uses some finite set of symbols as a set of primitives. The words are defined by finite sequences of symbols. Sentences are derived from finite sequences of words. Paragraphs are obtained from finite sequences of sentences, and so forth.

Similar approaches have been developed also for representing elements of other sets. For instance, the natural number can be represented by finite sequences of decimal digits.

Computations, like natural languages, are expected to deal with information in its most general form. Consequently, computations function as manipulators of integers, graphs, programs, and many other kinds of entities. However, in reality computations only manipulate strings of symbols that represent the objects. The previous discussion necessitates the following definitions.

Alphabets and Strings

A finite, nonempty ordered set will be called an *alphabet* if its elements are *symbols*, or *characters* (i.e., elements with "primitive" graphical representations). A finite sequence of symbols from a given alphabet will be called a *string* over the alphabet. A string that consists of a sequence $a_1,a_2,...,a_n$ of symbols will be denoted by the juxtaposition $a_1a_2...a_n$. Strings that have zero symbols, called *empty strings*, will be denoted by ε.

Example 1.1.1 $\Sigma_1=\{a,b,...,z\}$ and $\Sigma_2=\{0,1,...,9\}$ are alphabets. *abb* is a string over Σ_1, and 123 is a string over Σ_2. *ba*12 is not a string over Σ_1, because it contains symbols that are not in Σ_1. Similarly, 314159623... is not a string over Σ_2, because it is not a finite sequence. On the other hand, ε is a string over any alphabet.

The empty set $\varnothing$ is not an alphabet because it contains no element. The set of natural numbers is not an alphabet, because it is not finite. The union $\Sigma_1 \cup \Sigma_2$ is an alphabet only if an ordering is placed on its symbols. □

An alphabet of cardinality 2 is called a *binary alphabet*, and strings over a binary alphabet are called *binary strings*. Similarly, an alphabet of cardinality 1 is called a *unary alphabet*, and strings over a unary alphabet are called *unary strings*.

The *length* of a string α is denoted $|\alpha|$ and assumed to equal the number of symbols in the string.

Example 1.1.2 $\{0,1\}$ is a binary alphabet, and $\{1\}$ is a unary alphabet. 11 is a binary string over the alphabet $\{0,1\}$, and a unary string over the alphabet $\{1\}$.

11 is a string of length 2, $|\varepsilon|=0$, and $|01|+|1|=3$. □

The string consisting of a sequence α followed by a sequence β is denoted $\alpha\beta$. The string $\alpha\beta$ is called the *concatenation* of α and β. The notation α^i is used for the string obtained by concatenating i copies of the string α.

Example 1.1.3 The concatenation of the string 01 with the string 100 gives the string 01100. The concatenation $\varepsilon\alpha$ of ε with any string α, and the concatenation $\alpha\varepsilon$ of any string α with ε give the string α. In particular, $\varepsilon\varepsilon=\varepsilon$.

If $\alpha=01$, then $\alpha^0=\varepsilon$, $\alpha^1=01$, $\alpha^2=0101$, and $\alpha^3=010101$.

A string α is said to be a *substring* of a string β if $\beta=\gamma\alpha\rho$ for some γ and ρ. A substring α of a string β is said to be a *prefix* of β if $\beta=\alpha\rho$ for some ρ. The prefix is said to be a *proper prefix* of β if $\rho\neq\varepsilon$. A substring α of a string β is said to be a *suffix* of β if $\beta=\gamma\alpha$ for some γ. The suffix is said to be a *proper suffix* of β if $\gamma\neq\varepsilon$.

Example 1.1.4 ε, 0, 1, 01, 11, and 011 are the substrings of 011. ε, 0, and 01 are the proper prefixes of 011. ε, 1, and 11 are the proper suffixes of 011. 011 is a prefix and a suffix of 011.

If $\alpha=a_1...a_n$ for some symbols $a_1,...,a_n$ then $a_n...a_1$ is called the *reverse* of α, denoted α^{rev}. β is said to be a *permutation* of α if β can be obtained from α by reordering the symbols in α.

Example 1.1.5 Let α be the string 001. $\alpha^{rev}=100$. The strings 001, 010, and 100 are the permutations of α. □

The set of all the strings over an alphabet Σ will be denoted by Σ^*. Σ^+ will denote the set $\Sigma^* \setminus \{\varepsilon\}$.

Ordering of Strings

Searching is probably the most commonly applied operation on information.

Due to the importance of this operation, approaches for searching information and for organizing information to facilitate searching, receive special attention. Sequential search, binary search, insertion sort, quick sort, and merge sort are some examples of such approaches. These approaches rely in most cases on the existence of a relationship that defines an ordering of the entities in question.

A frequently used relationship for strings is the one that compares them alphabetically, as reflected by the ordering of names in telephone books. The relationship and ordering can be defined in the following manner.

Consider any alphabet Σ. A string α is said to be *alphabetically smaller* in Σ^* than a string β, or equivalently, β is said to be *alphabetically bigger* in Σ^* than α if α and β are in Σ^* and either of the following two cases holds.

a. α is a proper prefix of β.

b. For some γ in Σ^* and some a and b in Σ such that a precedes b in Σ, the string γa is a prefix of α and the string γb is a prefix of β.

An ordered subset of Σ^* is said to be *alphabetically ordered*, if β is not alphabetically smaller in Σ^* than α whenever α precedes β in the subset.

Example 1.1.6 Let Σ be the binary alphabet $\{0,1\}$. The string 01 is alphabetically smaller in Σ^* than the string 01100, because 01 is a proper prefix of 01100. On the other hand, 01100 is alphabetically smaller than 0111, because both strings agree in their first three symbols and the fourth symbol in 01100 is smaller than the fourth symbol in 0111.

The set $\{\varepsilon,0,00,000,001,01,010,011,1,10,100,101,11,110,111\}$, of those strings that have length not greater than 3, is given in alphabetical ordering. □

Alphabetical ordering is satisfactory for finite sets, because each string in such an ordered set can eventually be reached. For similar reasons, alphabetical ordering is also satisfactory for infinite sets of unary strings. However, in some other cases alphabetical ordering is not satisfactory because it can result in some strings being preceded by an unbounded number of strings. For instance, such is the case for the string 1 in the alphabetically ordered set $\{0,1\}^*$, that is, 1 is preceded by the strings 0, 00, 000,... This deficiency motivates the following definition of canonical ordering for strings. In canonical ordering each string is preceded by a finite number of strings.

A string α is said to be *canonically smaller* or *lexicographically smaller* in Σ^* than a string β, or equivalently, β is said to be *canonically bigger* or *lexicographically bigger* in Σ^* than α if either of the following two cases holds.

a. α is shorter than β.

b. α and β are of identical length but α is alphabetically smaller than β.

An ordered subset of Σ^* is said to be *canonically ordered* or *lexicographically ordered*, if β is not canonically smaller in Σ^* than α whenever α precedes β in the subset.

Example 1.1.7 Consider the alphabet $\Sigma=\{0,1\}$. The string 11 is canonically smaller in Σ^* than the string 000, because 11 is a shorter string than 000. On the other hand, 00 is canonically smaller than 11, because the strings are of equal length and 00 is alphabetically smaller than 11.

The set $\Sigma^*=\{\varepsilon,0,1,00,01,10,11,000,001,...\}$ is given in its canonical ordering. □

Representations

Given the preceding definitions of alphabets and strings, representations of information can be viewed as the mapping of objects into strings in accordance with some rules. That is, formally speaking, a *representation* or *encoding* over an alphabet Σ of a set D is a function f from D to 2^{Σ^*} that satisfies the following condition: $f(e_1)$ and $f(e_2)$ are disjoint nonempty sets for each pair of distinct elements e_1 and e_2 in D.

If Σ is a unary alphabet, then the representation is said to be a *unary representation.* If Σ is a binary alphabet, then the representation is said to be a *binary representation.*

In what follows each element in $f(e)$ will be referred to as a representation, or encoding, of e.

Example 1.1.8 f_1 is a binary representation over $\{0,1\}$ of the natural numbers if

$f_1(0)=\{0,00,000,0000,...\}$,

$f_1(1)=\{1,01,001,0001,...\}$,

$f_1(2)=\{10,010,0010,00010,...\}$,

$f_1(3)=\{11,011,0011,00011,...\}$, and

$f_1(4)=\{100,0100,00100,000100,...\}$, etc.

Similarly, f_2 is a binary representation over $\{0,1\}$ of the natural numbers if it assigns to the ith natural number the set consisting of the ith canonically smallest binary string. In such a case, $f_2(0)=\{\varepsilon\}$, $f_2(1)=\{0\}$, $f_2(2)=\{1\}$, $f_2(3)=\{00\}$, $f_2(4)=\{01\}$, $f_2(5)=\{10\}$, $f_2(6)=\{11\}$, $f_2(7)=\{000\}$, $f_2(8)=\{001\}$, $f_2(9)=\{010\}$,...

On the other hand, f_3 is a unary representation over $\{1\}$ of the natural numbers

if it assigns to the ith natural number the set consisting of the ith alphabetically (=canonically) smallest unary string. In such a case, $f_3(0)=\{\varepsilon\}$, $f_3(1)=\{1\}$, $f_3(2)=\{11\}$, $f_3(3)=\{111\}$, $f_3(4)=\{1111\}$,...; $f_3(i)=\{1^i\}$,...

The three representations f_1, f_2, and f_3 are illustrated in Figure 1.1.1.

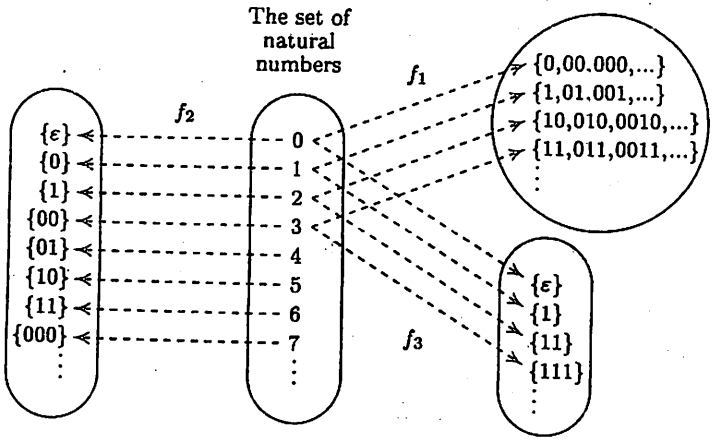


Fig. 1.1.1. Representations for the natural numbers

In the rest of the book, unless otherwise is stated, the function f_1 of Example 1.1.8 is assumed to be the binary representation of the natural numbers.

1.2 Formal Languages and Grammars

The universe of strings is a useful medium for the representation of information as long as there exists a function that provides the interpretation for the information carried by the strings. An interpretation is just the inverse of the mapping that a representation provides, that is, an *interpretation* is a function g from Σ^* to D for some alphabet Σ and some set D. The string 111, for instance, can be interpreted as the number one hundred and eleven represented by a decimal string, as the number seven represented by a binary string, and as the number three represented by a unary string.

The parties communicating a piece of information do the representing and interpreting. The representation is provided by the sender, and the interpretation is provided by the receiver. The process is the same no matter whether the parties are human beings or programs. Consequently, from the point of view of the parties involved, a language can be just a collection of strings because the parties embed the representation and interpretation functions in themselves.

Languages

In general, if Σ is an alphabet and L is a subset of Σ^*, then L is said to be a *language* over Σ, or simply a language if Σ is understood. Each element of L is said to be a *sentence* or a *word* or a *string* of the language.

Example 1.2.1 $\{0,11,001\}$, $\{\varepsilon,10\}$, and $\{0,1\}^*$ are subsets of $\{0,1\}^*$, and so they are languages over the alphabet $\{0,1\}$.

The empty set $\varnothing$ and the set $\{\varepsilon\}$ are languages over every alphabet. $\varnothing$ is a language that contains no string. $\{\varepsilon\}$ is a language that contains just the empty string. □

The *union* of two languages L_1 and L_2, denoted $L_1 \cup L_2$, refers to the language that consists of all the strings that are either in L_1 or in L_2, that is, to $\{x \mid x \text{ is in } L_1 \text{ or } x \text{ is in } L_2\}$. The *intersection* of L_1 and L_2, denoted $L_1 \cap L_2$, refers to the language that consists of all the strings that are both in L_1 and L_2, that is, to $\{x \mid x \text{ is in } L_1 \text{ and in } L_2\}$. The *complementation* of a language L over Σ, or just the complementation of L when Σ is understood, denoted $\overline{L}$, refers to the language that consists of all the strings over Σ that are not in L, that is, to $\{x \mid x \text{ is in } \Sigma^* \text{ but not in } L\}$.

Example 1.2.2 Consider the languages $L_1=\{\varepsilon,0,1\}$ and $L_2=\{\varepsilon,01,11\}$. The union of these languages is $L_1 \cup L_2=\{\varepsilon,0,1,01,11\}$, their intersection is $L_1 \cap L_2=\{\varepsilon\}$, and the complementation of L_1 is $\overline{L_1}=\{00,01,10,11,000,001,...\}$.

$\varnothing \cup L = L$ for each language L. Similarly, $\varnothing \cap L = \varnothing$ for each language L. On the other hand, $\overline{\varnothing}=\Sigma^*$ and $\overline{\Sigma^*}=\varnothing$ for each alphabet Σ. □

The *difference* of L_1 and L_2, denoted $L_1 \setminus L_2$, refers to the language that consists of all the strings that are in L_1 but not in L_2, that is, to

$$\{x \mid x \text{ is in } L_1 \text{ but not in } L_2\}.$$

The *cross product* of L_1 and L_2, denoted $L_1 \times L_2$, refers to the set of all the pairs (x,y) of strings such that x is in L_1 and y is in L_2, that is, to the relation $\{(x,y) \mid x \text{ is in } L_1 \text{ and } y \text{ is in } L_2\}$. The *composition* of L_1 with L_2, denoted L_1L_2, refers to the language $\{xy \mid x \text{ is in } L_1 \text{ and } y \text{ is in } L_2\}$.

Example 1.2.3 If $L_1=\{\varepsilon,1,01,11\}$ and $L_2=\{1,01,101\}$ then $L_1 \setminus L_2=\{\varepsilon,11\}$ and $L_2 \setminus L_1=\{101\}$.

On the other hand, if $L_1=\{\varepsilon,0,1\}$ and $L_2=\{01,11\}$, then the cross product of these languages is $L_1 \times L_2=\{(\varepsilon,01),(\varepsilon,11),(0,01),(0,11),(1,01),(1,11)\}$, and their composition is $L_1L_2=\{01,11,001,011,101,111\}$.

$L \setminus \varnothing = L$, $\varnothing \setminus L = \varnothing$, $\varnothing L = \varnothing$, and $\{\varepsilon\}L = L$ for each language L. □

L^i will also be used to denote the composing of i copies of a language L, where

L^0 is defined as $\{\varepsilon\}$. The set $L^0 \cup L^1 \cup L^2 \cup L^3 \cup \ldots$, called the *Kleene closure* or just the *closure* of L, will be denoted by L^*. The set $L^1 \cup L^2 \cup L^3 \ldots$, called the *positive closure* of L, will be denoted by L^+.

L^i consists of those strings that can be obtained by concatenating i strings from L. L^* consists of those strings that can be obtained by concatenating an arbitrary number of strings from L.

Example 1.2.4 Consider the pair of languages $L_1 = \{\varepsilon,0,1\}$ and $L_2 = \{01,11\}$. For these languages

$L_1^2 = \{\varepsilon,0,1,00,01,10,11\}$, and

$L_2^3 = \{010101,010111,011101,011111,110101,110111,111101,111111\}$.

In addition, ε is in L_1^*, in L_1^+, and in L_2^* but not in L_2^+. □

The operations above apply in a similar way to relations in $\Sigma^* \times \Delta^*$, when Σ and Δ are alphabets. Specifically, the *union* of the relations R_1 and R_2, denoted $R_1 \cup R_2$, is the relation $\{(x,y) \mid (x,y)$ *is in* R_1 *or in* $R_2\}$. The *intersection* of R_1 and R_2, denoted $R_1 \cap R_2$, is the relation $\{(x,y) \mid (x,y)$ *is in* R_1 *and in* $R_2\}$. The *composition* of R_1 with R_2, denoted $R_1 R_2$, is the relation

$$\{(x_1x_2,y_1y_2) \mid (x_1,y_1) \text{ is in } R_1 \text{ and } (x_2,y_2) \text{ is in } R_2\}.$$

Example 1.2.5 Consider the relations

$$R_1 = \{(\varepsilon,0),(10,1)\}, \text{ and } R_2 = \{(1,\varepsilon),(0,01)\}.$$

For these relations

$R_1 \cup R_2 = \{(\varepsilon,0),(10,1),(1,\varepsilon),(0,01)\}$,

$R_1 \cap R_2 = \varnothing$,

$R_1 R_2 = \{(1,0),(0,001),(101,1),(100,101)\}$, and

$R_2 R_1 = \{(1,0),(110,1),(0,010),(010,011)\}$. □

The *complementation* of a relation R in $\Sigma^* \times \Delta^*$, or just the complementation of R when Σ and Δ are understood, denoted $\overline{R}$, is the relation

$$\{(x,y) \mid (x,y) \text{ is in } \Sigma^* \times \Delta^* \text{ but not in } R\}.$$

The *inverse* of R, denoted R^{-1}, is the relation $\{(y,x) \mid (x,y)$ *is in* $R\}$. $R^0 = \{(\varepsilon,\varepsilon)\}$. $R^i = R^{i-1}R$ for $i \geq 1$.

Example 1.2.6 If R is the relation $\{(\varepsilon,\varepsilon),(\varepsilon,01)\}$, then $R^{-1} = \{(\varepsilon,\varepsilon),(01,\varepsilon)\}$, $R^0 = \{(\varepsilon,\varepsilon)\}$, and $R^2 = \{(\varepsilon,\varepsilon),(\varepsilon,01),(\varepsilon,0101)\}$. □

A language that can be defined by a formal system, that is, by a system that has a finite number of axioms and a finite number of inference rules, is said to be a *formal language*.

Grammars

It is often convenient to specify languages in terms of grammars. The advantage in doing so arises mainly from the usage of a small number of rules for describing a language with a large number of sentences. For instance, the possibility that an English sentence consists of a subject phrase followed by a predicate phrase can be expressed by a grammatical rule of the form

$$\langle sentence \rangle \rightarrow \langle subject \rangle \langle predicate \rangle.$$

(The names in angular brackets are assumed to belong to the grammar meta-language.) Similarly, the possibility that the subject phrase consists of a noun phrase can be expressed by a grammatical rule of the form

$$\langle subject \rangle \rightarrow \langle noun \rangle.$$

In a similar manner it can also be deduced that "Mary sang a song" is a possible sentence in the language described by the following grammatical rules.

⟨sentence⟩ → ⟨subject⟩⟨predicate⟩
⟨subject⟩ → ⟨noun⟩
⟨predicate⟩ → ⟨verb⟩⟨article⟩⟨noun⟩
⟨noun⟩ → ⟨name⟩
⟨noun⟩ → ⟨string⟩
⟨name⟩ → ⟨u_character⟩⟨string⟩
⟨string⟩ → ⟨string⟩⟨character⟩
⟨string⟩ → ⟨character⟩
⟨character⟩ → a
⋮
⟨character⟩ → z
⟨u_character⟩ → A
⋮
⟨u_character⟩ → Z
⟨verb⟩ → sang
⟨article⟩ → a

The grammatical rules above also allow English sentences of the form "Mary sang a song" for other names besides Mary. On the other hand, the rules imply non-English sentences like "Mary sang a Mary," and do not allow English

sentences like "Mary read a song." Therefore, the set of grammatical rules above consists of an incomplete grammatical system for specifying the English language.

For the investigation conducted here it is sufficient to consider only grammars that consist of finite sets of grammatical rules of the previous form. Such grammars are called Type 0 grammars, or phrase structure grammars, and the formal languages that they generate are called Type 0 languages.

Strictly speaking, each *Type 0 grammar* G is defined as a mathematical system consisting of a quadruple $\langle N,\Sigma,P,S\rangle$, where

- N is an alphabet, whose elements are called *nonterminal* symbols.
- Σ is an alphabet disjoint from N, whose elements are called *terminal* symbols.
- P is a relation of finite cardinality on $(N\cup\Sigma)^*$, whose elements are called *production rules*. Moreover, each production rule (α,β) in P, denoted $\alpha\rightarrow\beta$, must have at least one nonterminal symbol in α. In each such production rule, α is said to be the *left-hand side* of the production rule, and β is said to be the *right-hand side* of the production rule.
- S is a symbol in N called the *start*, or *sentence*, symbol.

Example 1.2.7 $\langle N,\Sigma,P,S\rangle$ is a Type 0 grammar if $N=\{S\}$, $\Sigma=\{a,b\}$, and $P=\{S\rightarrow aSb, S\rightarrow\varepsilon\}$. By definition, the grammar has a single nonterminal symbol S, two terminal symbols a and b, and two production rules $S\rightarrow aSb$ and $S\rightarrow\varepsilon$. Both production rules have a left-hand side that consists only of the nonterminal symbol S. The right-hand side of the first production rule is aSb, and the right-hand side of the second production rule is ε.

$\langle N_1,\Sigma_1,P_1,S\rangle$ is not a grammar if N_1 is the set of natural numbers, or Σ_1 is empty, because N_1 and Σ_1 have to be alphabets.

If $N_2=\{S\}$, $\Sigma_2=\{a,b\}$, and $P_2=\{S\rightarrow aSb, S\rightarrow\varepsilon, ab\rightarrow S\}$ then $\langle N_2,\Sigma_2,P_2,S\rangle$ is not a grammar, because $ab\rightarrow S$ does not satisfy the requirement that each production rule must contain at least one nonterminal symbol on the left-hand side. □

In general, the nonterminal symbols of a Type 0 grammar are denoted by S and by the first uppercase letters in the English alphabet A, B, C, D, and E. The start symbol is denoted by S. The terminal symbols are denoted by digits and by the first lowercase letters in the English alphabet a, b, c, d, and e. Symbols of insignificant nature are denoted by X, Y, and Z. Strings of terminal symbols are denoted by the last lowercase English characters u, v, w, x, y, and z. Strings that may consist of both terminal and nonterminal symbols are denoted by the first lowercase Greek symbols α, β, and γ. In addition, for

convenience, sequences of production rules of the form

$$\begin{aligned}\alpha &\rightarrow \beta_1\\ \alpha &\rightarrow \beta_2\\ &\vdots\\ \alpha &\rightarrow \beta_n\end{aligned}$$

are denoted as

$$\begin{aligned}\alpha &\rightarrow \beta_1\\ &\rightarrow \beta_2\\ &\vdots\\ &\rightarrow \beta_n\end{aligned}$$

Example 1.2.8 $\langle N,\Sigma,P,S\rangle$ is a Type 0 grammar if $N=\{S,B\}$, $\Sigma=\{a,b,c\}$, and P consists of the following production rules.

$$\begin{aligned}S &\rightarrow aBSc\\ &\rightarrow abc\\ &\rightarrow \varepsilon\\ Ba &\rightarrow aB\\ Bb &\rightarrow bb\end{aligned}$$

The nonterminal symbol S is the left-hand side of the first three production rules. Ba is the left-hand side of the fourth production rule. Bb is the left-hand side of the fifth production rule.

The right-hand side $aBSc$ of the first production rule contains both terminal and nonterminal symbols. The right-hand side abc of the second production rule contains only terminal symbols. Except for the trivial case of the right-hand side ε of the third production rule, none of the right-hand sides of the production rules consists only of nonterminal symbols, even though they are allowed to be of such a form. □

Derivations

Grammars generate languages by repeatedly modifying given strings. Each modification of a string is in accordance with some production rule of the grammar in question $G=\langle N,\Sigma,P,S\rangle$. A modification to a string γ in accordance with production rule $\alpha\rightarrow\beta$ is derived by replacing a substring α in γ by β.

In general, a string γ is said to directly derive a string γ' if γ' can be obtained from γ by a single modification. Similarly, a string γ is said to derive γ' if γ' can be obtained from γ by a sequence of an arbitrary number of direct derivations.

Formally, a string γ is said to *directly derive* in G a string γ', denoted $\gamma \Rightarrow_G \gamma'$, if γ' can be obtained from γ by replacing a substring α with β, where $\alpha \rightarrow \beta$ is a production rule in G. That is, if $\gamma = \rho\alpha\delta$ and $\gamma' = \rho\beta\delta$ for some strings α, β, ρ, and δ such that $\alpha \rightarrow \beta$ is a production rule in G.

Example 1.2.9 If G is the grammar $\langle N, \Sigma, P, S \rangle$ in Example 1.2.7, then both ε and aSb are directly derivable from S. Similarly, both ab and a^2Sb^2 are directly derivable from aSb. ε is directly derivable from S, and ab is directly derivable from aSb, in accordance with the production rule $S \rightarrow \varepsilon$. aSb is directly derivable from S, and a^2Sb^2 is directly derivable from aSb, in accordance with the production rule $S \rightarrow aSb$.

On the other hand, if G is the grammar $\langle N, \Sigma, P, S \rangle$ of Example 1.2.8, then $aBaBabccc \Rightarrow_G aaBBabccc$ and $aBaBabccc \Rightarrow_G aBaaBbccc$ in accordance with the production rule $Ba \rightarrow aB$. Moreover, no other string is directly derivable from $aBaBabccc$ in G. □

γ is said to *derive* γ' in G, denoted $\gamma \Rightarrow_G^* \gamma'$, if $\gamma_0 \Rightarrow_G \ldots \Rightarrow_G \gamma_n'$ for some $\gamma_0, \ldots, \gamma_n$ such that $\gamma_0 = \gamma$ and $\gamma_n = \gamma'$. In such a case, the sequence $\gamma_0 \Rightarrow_G \ldots \Rightarrow_G \gamma_n$ is said to be a *derivation* of γ from γ' whose *length* is equal to n. $\gamma_0, \ldots, \gamma_n$ are said to be *sentential forms*, if $\gamma_0 = S$. A sentential form that contains no terminal symbols is said to be a *sentence*.

Example 1.2.10 If G is the grammar of Example 1.2.7, then a^4Sb^4 has a derivation from S. The derivation $S \Rightarrow_G^* a^4Sb^4$ has length 4, and it has the form $S \Rightarrow_G aSb \Rightarrow_G a^2Sb^2 \Rightarrow_G a^3Sb^3 \Rightarrow_G a^4Sb^4$. □

A string is assumed to be in the language that the grammar G generates if and only if it is a string of terminal symbols that is derivable from the starting symbol. The language that is *generated* by G, denoted $L(G)$, is the set of all the strings of terminal symbols that can be derived from the start symbol, that is, the set $\{w \mid w \text{ is in } \Sigma^*, \text{ and } S \Rightarrow_G^* w\}$. Each string in the language $L(G)$ is said to be generated by G.

Example 1.2.11 Consider the grammar G of Example 1.2.7. ε is in the language that G generates because of the existence of the derivation $S \Rightarrow_G \varepsilon$. ab is in the language that G generates, because of the existence of the derivation $S \Rightarrow_G aSb \Rightarrow_G ab$. a^2b^2 is in the language that G generates, because of the existence of the derivation $S \Rightarrow_G aSb \Rightarrow_G a^2Sb^2 \Rightarrow_G a^2b^2$.

The language $L(G)$ that G generates consists of all the strings of the form $a \ldots ab \ldots b$ in which the number of a's is equal to the number of b's, that is, $L(G) = \{a^ib^i \mid i \text{ is a natural number}\}$.

aSb is not in $L(G)$ because it contains a nonterminal symbol. a^2b is not in $L(G)$ because it cannot be derived from S in G. □

In what follows, the notations $\gamma \Rightarrow \gamma'$ and $\gamma \Rightarrow^* \gamma'$ are used instead of $\gamma \Rightarrow_G \gamma'$ and $\gamma \Rightarrow_G^* \gamma'$, respectively, when G is understood. In addition, Type 0 grammars are referred to simply as grammars, and Type 0 languages are referred to simply as languages, when no confusion arises.

Example 1.2.12 If G is the grammar of Example 1.2.8, then the following is a derivation for $a^3b^3c^3$. The underlined and the overlined substrings are the left- and the right-hand sides, respectively, of those production rules used in the derivation.

$$\begin{aligned}
\underline{S} &\Rightarrow \overline{aB\underline{S}c} \\
&\Rightarrow aBa\overline{B\underline{S}c}c \\
&\Rightarrow aBa\underline{B}\overline{abc}cc \\
&\Rightarrow aBaa\overline{\underline{B}bc}cc \\
&\Rightarrow a\underline{B}aa\overline{bb}ccc \\
&\Rightarrow aa\overline{\underline{B}a}bbccc \\
&\Rightarrow aaa\overline{\underline{B}b}bccc \\
&\Rightarrow aaa\overline{bbb}ccc
\end{aligned}$$

The language generated by the grammar G consists of all the strings of the form $a \ldots ab \ldots bc \ldots c$ in which there are equal numbers of a's, b's, and c's, that is, $L(G) = \{a^ib^ic^i \mid i \text{ is a natural number}\}$.

The first two production rules in G are used for generating sentential forms that have the pattern $aBaB \ldots aBabc \ldots c$. In each such sentential form the number of a's is equal to the number of c's and is greater by 1 than the number of B's.

The production rule $Ba \rightarrow aB$ is used for transporting the B's rightward in the sentential forms. The production rule $Bb \rightarrow bb$ is used for replacing the B's by b's, upon reaching their appropriate positions. □

Derivation Graphs

Derivations of sentential forms in Type 0 grammars can be displayed by *derivation*, or *parse*, *graphs*. Each derivation graph is a rooted, ordered, acyclic, directed graph whose nodes are labeled. The label of each node is either a nonterminal symbol, a terminal symbol, or an empty string. The derivation graph that corresponds to a derivation $S \Rightarrow \gamma_1 \Rightarrow \ldots \Rightarrow \gamma_n$ is defined inductively in the following manner.

a. The derivation graph D_0 that corresponds to S consists of a single node labeled by the start symbol S.

b. If $\alpha \rightarrow \beta$ is the production rule used in the direct derivation $\gamma_i \Rightarrow \gamma_{i+1}, 0 \leq i < n$ and $\gamma_0 = S$, then the derivation graph D_{i+1} that corresponds to

$\gamma_0 \Rightarrow ... \Rightarrow \gamma_{i+1}$ is obtained from D_i by the addition of $\max(|\beta|,1)$ new nodes. The new nodes are labeled by the characters of β, and are assigned as common successors to each of the nodes in D_i that corresponds to a character in α. Consequently, the leaves of the derivation graph D_{i+1} are labeled by γ_{i+1}.

Derivation graphs are also called *derivation trees* or *parse trees* when the directed graphs are trees.

Example 1.2.13 Figure 1.2.1(a) provides examples of derivation trees for derivations in the grammar of Example 1.2.7. Figure 1.2.1(b) provides examples of derivation graphs for derivations in the grammar of Example 1.2.8. □

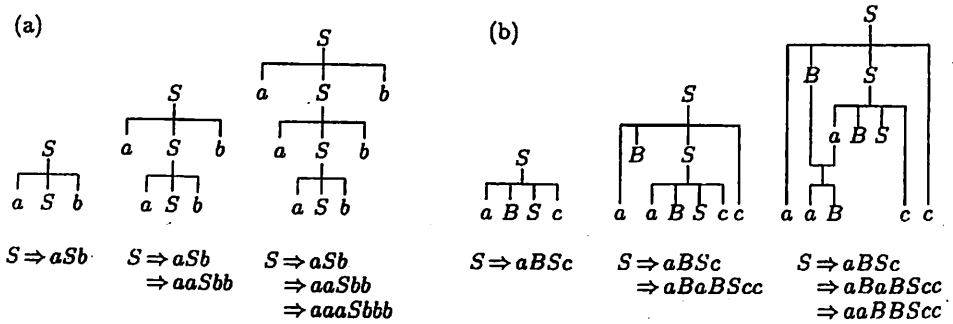


Fig. 1.2.1. (a) Derivation trees. (b) Derivation graphs

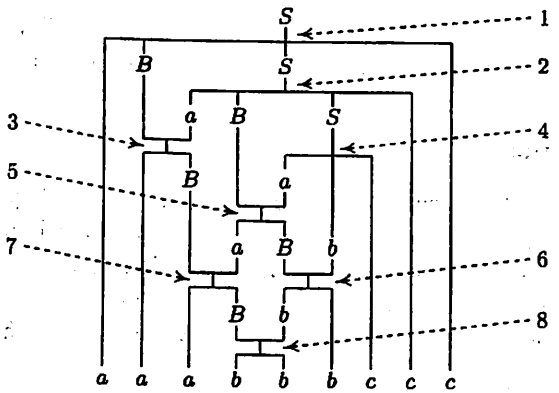


Fig. 1.2.2. A derivation graph with oredering of the usage of production rules indicated with arrows

Leftmost Derivations

A derivation $\gamma_0 \Rightarrow ... \Rightarrow \gamma_n$ is said to be a *leftmost derivation* if α_1 is replaced before α_2 in the derivation whenever the following two conditions hold.

a. α_1 appears to the left of α_2 in γ_i, $0 \leqslant i < n$.

b. α_1 and α_2 are replaced during the derivation in accordance with some production rules of the form $\alpha_1 \rightarrow \beta_1$ and $\alpha_2 \rightarrow \beta_2$, respectively.

Example 1.2.14 The derivation graph in Figure 1.2.2 indicates the order in which the production rules are used in the derivation of $a^3b^3c^3$ in Example 1.2.12. The substring $\alpha_1 = aB$ that is replaced in the seventh step of the derivation is in the same sentential form as the substring $\alpha_2 = Bb$ that is replaced in the sixth step of the derivation. The derivation is not a leftmost derivation because α_1 appears to the left of α_2 while it is being replaced after α_2.

On the other hand, the following derivation is a leftmost derivation for $a^3b^3c^3$ in G. The order in which the production rules are used is similar to that indicated in Figure 1.2.2. The only difference is that the indices 6 and 7 should be interchanged.

$$\begin{aligned}
\underline{S} &\Rightarrow \overline{aB\underline{Sc}} \\
&\Rightarrow \underline{aB\overline{a}\overline{BSc}}c \\
&\Rightarrow aa\overline{\overline{B}B\underline{Sc}}c \\
&\Rightarrow aaB\underline{B\overline{a}\overline{bc}}cc \\
&\Rightarrow aa\underline{B\overline{a}}\overline{\overline{B}bc}cc \\
&\Rightarrow aaa\overline{\overline{B}}\underline{Bb}cc \\
&\Rightarrow aaaB\overline{\overline{b}\overline{b}}ccc \\
&\Rightarrow aaa\overline{\overline{b}\overline{b}bc}cc
\end{aligned}$$

Hierarchy of Grammars

The following classes of grammars are obtained by gradually increasing the restrictions that the production rules have to obey.

A *Type 1 grammar* is a Type 0 grammar $\langle N,\Sigma,P,S\rangle$ that satisfies the following two conditions.

a. Each production rule $\alpha \rightarrow \beta$ in P satisfies $|\alpha| \leqslant |\beta|$ if it is not of the form $S \rightarrow \varepsilon$.

b. If $S \rightarrow \varepsilon$ is in P, then S does not appear in the right-hand side of any production rule.

A language is said to be a *Type 1 language* if there exists a Type 1 grammar that generates the language.

Example 1.2.15 The grammar of Example 1.2.8 is not a Type 1 grammar, because it does not satisfy condition (b). The grammar can be modified to be of Type 1 by replacing its production rules with the following ones. E is assumed to be a new nonterminal symbol.

$$\begin{aligned} S &\to E \\ &\to \varepsilon \\ E &\to aBEc \\ &\to abc \\ Ba &\to aB \\ Bb &\to bb \end{aligned}$$

An addition to the modified grammar of a production rule of the form $Bb \to b$ will result in a non-Type 1 grammar, because of a violation to condition (a). □

A *Type 2 grammar* is a Type 1 grammar in which each production rule $\alpha \to \beta$ satisfies $|\alpha| = 1$, that is, α is a nonterminal symbol. A language is said to be a *Type 2 language* if there exists a Type 2 grammar that generates the language.

Example 1.2.16 The grammar of Example 1.2.7 is not a Type 1 grammar, and therefore also not a Type 2 grammar. The grammar can be modified to be a Type 2 grammar, by replacing its production rules with the following ones. E is assumed to be a new nonterminal symbol.

$$\begin{aligned} S &\to \varepsilon \\ &\to E \\ E &\to aEb \\ &\to ab \end{aligned}$$

An addition of a production rule of the form $aE \to EaE$ to the grammar will result in a non-Type 2 grammar. □

A *Type 3 grammar* is a Type 2 grammar $\langle N, \Sigma, P, S \rangle$ in which each of the production rules $\alpha \to \beta$, which is not of the form $S \to \varepsilon$, satisfies one of the following conditions.

a. β is a terminal symbol.

b. β is a terminal symbol followed by a nonterminal symbol.

A language is said to be a *Type 3 language* if there exists a Type 3 grammar that generates the language.

Example 1.2.17 The grammar $\langle N, \Sigma, P, S \rangle$, which has the following production rules, is a Type 3.

$$\begin{aligned} S &\to \varepsilon \\ &\to aA \\ &\to bB \\ &\to b \\ A &\to bB \\ &\to b \\ B &\to aA \\ &\to bB \\ &\to b \end{aligned}$$

An addition of a production rule of the form $A \to Ba$, or of the form $B \to bb$, to the grammar will result in a non-Type 3 grammar. □

Figure 1.2.3 illustrates the hierarchy of the different types of grammars.

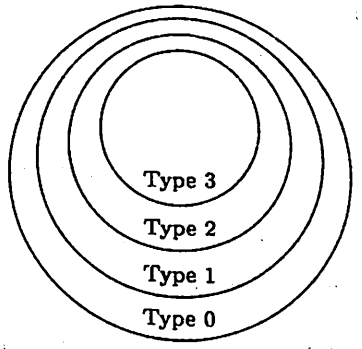


Fig. 1.2.3. Hierarchy of grammars

1.3 Programs

Our deep dependency on the processing of information brought about the deployment of programs in an ever increasing array of applications. Programs can be found at home, at work, and in businesses, libraries, hospitals, and schools. They are used for learning, playing games, typesetting, directing telephone calls, providing medical diagnostics, forecasting weather, flying airplanes, and for many other purposes.

To facilitate the task of writing programs for the multitude of different applications, numerous programming languages have been developed. The diversity of programming languages reflects the different interpretations that can be given to information. However, from the perspective of their power to express computations, there is very little difference among them. Consequently, different programming languages can be used in the study of programs.

The study of programs can benefit, however, from fixing the programming language in use. This enables a unified discussion about programs. The choice, however, must be for a language that is general enough to be relevant to all programs but primitive enough to simplify the discussion.

Choice of a Programming Language

Here, a *program* is defined as a finite sequence of instructions over some domain D. The domain D, called the *domain of the variables*, is assumed to be a set of elements with a distinguished element, called the *initial value of the variables*. Each of the elements in D is assumed to be a possible assignment of a value to the variables of the program. The sequence of instructions is assumed to consist of instructions of the following form.

a. Read instructions of the form
read x
where x is a variable.

b. Write instructions of the form
write x
where x is a variable.

c. Deterministic assignment instructions of the form
$y := f(x_1,...,x_m)$
where $x_1,...,x_m$, and y are variables, and f is a function from D^m to D.

d. Conditional if instructions of the form
if $Q(x_1,...,x_m)$ **then** I
where I is an instruction, $x_1,...,x_m$ are variables, and Q is a predicate from D^m to {**false**,**true**}.

e. Deterministic looping instructions of the form
do α **until** $Q(x_1,...,x_m)$
where α is a nonempty sequence of instructions, $x_1,...,x_m$ are variables, and Q is a predicate from D^m to {**false**,**true**}.

f. Conditional accept instructions of the form
if *eof* **then accept**

g. Reject instructions of the form
reject

h. Nondeterministic assignment instructions of the form
$x :=?$
where x is a variable.

i. Nondeterministic looping instructions of the form
do α_1
or α_2

⋮
or α_k
until $Q(x_1,...,x_m)$
where $k \geq 2$, each of $\alpha_1,...,\alpha_k$ is a nonempty sequence of instructions, $x_1,...,x_m$ are variables, and Q is a predicate from D^m to {**false**,**true**}.

In each program the domain D of the variables is assumed to have a representation over some alphabet. For instance, D can be the set of natural numbers, the set of integers, and any finite set of elements. The functions f and predicates Q are assumed to be from a given "built-in" set of computable functions and predicates (see Section 1.4 and Church's thesis in Section 4.1).

In what follows, the domains of the variables will not be explicitly noted when their nature is of little significance. In addition, expressions in infix notations will be used for specifying functions and predicates.

Programs without nondeterministic instructions are called *deterministic* programs, and programs with nondeterministic instructions are called *nondeterministic* programs.

(a)
```
read x
y := 0
z := 1
do
    y := y+1
    z := z+1
until z = x
read y
if eof then accept
reject
```

(b)
```
do
    read x
or
    y :=?
    write y
until y = x
if eof then accept
```

Fig. 1.3.1 (a) A deterministic program. (b) A nondeterministic program

Example 1.3.1 The program P_1 in Figure 1.3.1(a) is an example of a deterministic program, and the program P_2 in Figure 1.3.1(b) is an example of a nondeterministic program. The set of natural numbers is assumed for the domains of the variables, with 0 as initial value.

The program P_1 uses three variables, namely, x, y, and z. There are two functions in this program. The constant function $f_1() = 0$, and the unary function $f_2(n) = n+1$ of addition by one. The looping instruction uses the binary predicate of equality.

The program P_2 uses two nondeterministic instructions. One of the nondeterministic instructions is an assignment instruction of the form "$y :=?$"; the other is a looping instruction of the form "**do... or... until...** " □

An *input* of a given program is a sequence of elements from the domain of the variables of the program. Each element in an input of a program is called an *input value*.

Example 1.3.2 The programs of Example 1.3.1 (see Figure 1.3.1) can have any input that is a finite sequence of natural numbers. An input of the form "1,2,3,4" consists of four input values, and an input of the form " " contains no input value.

The sequence "1,2,3,..." cannot be an input for the programs because it is not a finite sequence. □

An *execution sequence* of a given program is an execution on a given input of the instructions according to their semantics. The instructions are executed consecutively, starting with the first instruction. The variables initially hold the initial value of the variables.

Deterministic Programs

Deterministic programs have the property that no matter how many times they are executed on a given input, the executions are always in exactly the same manner. Each instruction of a deterministic program fully specifies the operations to be performed. In contrast, nondeterministic instructions provide only partial specifications for the actions.

An execution of a read instruction read x reads the next input value to x. An execution of a write instruction write x writes the value of x.

The deterministic assignment instructions and the conditional if instructions have the conventional semantics.

An execution of a deterministic looping instruction **do** α **until** $Q(x_1,...,x_m)$ consists of repeatedly executing α and checking the value of $Q(x_1,...,x_m)$. The execution of the looping instruction is terminated upon detecting that the predicate $Q(x_1,...,x_m)$ has the value `true`. If $Q(x_1,...,x_m)$ is the constant `true`, then only one iteration is executed. On the other hand, if $Q(x_1,...,x_m)$ is the constant `false`, then the looping goes on forever, unless the execution terminates in α.

A conditional accept instruction causes an execution sequence to halt if executed after all the input is consumed, that is, after reaching the end of input file (*eof* for short). Otherwise the execution of the instruction causes the execution sequence to continue at the code following the instruction. Similarly, an execution sequence also halts upon executing a reject instruction, trying to read beyond the end of the input, trying to transfer the control beyond the end of the program, or trying to compute a value not in the domain of the variables (e.g, trying to divide by 0).

(a)
```
do
    if eof then accept
    read value
    write value
until false
```

(b)
```
do
    read value
    write value
until value<0
if eof then accept
```

Fig 1.3.2 Two deterministic programs

Example 1.3.3 Consider the two programs in Figure 1.3.2. Assume that the programs have the set of integers for the domains of their variables, with 0 as initial value.

For each input the program in Figure 1.3.2(a) has one execution sequence. In each execution sequence the program provides an output that is equal to the input. All the execution sequences of the program terminate due to the execution of the conditional accept instruction.

On input "1,2" the execution sequence repeatedly executes for three times the body of the deterministic looping instruction. During the first iteration, the execution sequence determines that the predicate *eof* has the value `false`. Consequently, the execution sequence ignores the accept command and continues by reading the value 1 and writing it out. During the second iteration the execution sequence verifies again that the end of the input has not been reached yet, and then the execution sequence reads the input value 2 and writes it out. During the third iteration, the execution sequence terminates due to the accept command, after determining a `true` value for the predicate *eof*.

The execution sequences of the program in Figure 1.3.2(b) halt due to the conditional accept instruction, only on inputs that end with a negative value and have no negative values elsewhere (e.g, the input "1,2,−3"). On inputs that contain no negative values at all, the execution sequences of the program halt due to trying to read beyond the end of the input (e.g, on input "1,2,3"). On inputs that have negative values before their end, the execution sequences of the program halt due to the transfer of control beyond the end of the program (e.g, on input "−1,2,−3"). □

Intuitively, an accept can be viewed as a halt command that signals a successful completion of a program execution, where the accept can be executed only after the end of the input is reached. Similarly, a reject can be viewed as a halt instruction that signals an unsuccessful completion of a program execution.

The requirement that the accept commands be executed only after reading all the input values should cause no problem, because each program can be modified to satisfy this condition. Moreover, such a constraint seems to be natural, because it forces each program to check all its input values before signaling a success by an accept command. Similarly, the requirement that an

execution sequence must halt upon trying to read beyond the end of an input seems to be natural. It should not matter whether the reading is due to a read instruction or to checking for the *eof* predicate.

It should be noted that the predicates $Q(x_1,...,x_m)$ in the conditional if instructions and in the looping instructions cannot be of the form eof. The predicates are defined just in terms of the values of the variables $x_1,...,x_m$, not in terms of the input.

Computations

Programs use finite sequences of instructions for describing sets of infinite numbers of computations. The descriptions of the computations are obtained by "unrolling" the sequences of instructions into execution sequences. In the case of deterministic programs, each execution sequence provides a description for a computation. On the other hand, as it will be seen below, in the case of nondeterministic programs some execution sequences might be considered as computations, whereas others might be considered noncomputations. To delineate this distinction we need the following definitions.

An execution sequence is said to be an *accepting computation* if it terminates due to an accept command. An execution sequence is said to be a *nonaccepting computation* or a *rejecting computation* if it is on input that has no accepting computations. An execution sequence is said to be a *computation* if it is an accepting computation or a nonaccepting computation.

A computation is said to be a *halting computation* if it is finite.

Example 1.3.4 Consider the program in Figure 1.3.3. Assume that the domain of the variables is the set of integers, with 0 as initial value.

On an input that consists of a single, even, positive integer, the program has an execution sequence that is an accepting computation (e.g, on input "4").

```
read value
do
    write value
    value := value − 2
until value = 0
if eof then accept
```

Fig 1.3.3 A deterministic program

On an input that consists of more than one value and that starts with an even positive integer, the program has a halting execution sequence that is a nonaccepting computation (e.g, on input "4,3,2").

On the rest of the inputs the program has nonhalting execution sequences that are nonaccepting computations (e.g, on input "1"). □

An input is said to be *accepted*, or *recognized*, by a program if the program has an accepting computation on such an input. Otherwise the input is said to be *not accepted*, or *rejected*, by the program.

A program is said to have an *output* y on input x if it has an accepting computation on x with output y. The outputs of the nonaccepting computations are considered to be undefined, even though such computations may execute write instructions.

Example 1.3.5 The program in Example 1.3.4 (see Figure 1.3.3) accepts the inputs "2", "4", "6",... On input "6" the program has the output "6,4,2", and on input "2" the program has the output "2".

The program does not accept the inputs "0", "1", and "4,2". For these inputs the program has no output, that is, the output is undefined. □

A computation is said to be a *nondeterministic computation* if it involves the execution of a nondeterministic instruction. Otherwise the computation is said to be a *deterministic computation*.

Nondeterministic Programs

Different objectives create the need for nondeterministic instructions in programming languages. One of the objectives is to allow the programs to deal with problems that may have more than one solution. In such a case, nondeterministic instructions provide a natural method of selection (see, e.g., Example 1.3.6 below). Another objective is to simplify the task of programming (see, e.g., Example 1.3.9 below). Still another objective is to provide tools for identifying difficult problems (see Chapter 5) and for studying restricted classes of programs (see Chapter 2 and Chapter 3).

Implementation considerations should not bother the reader at this point. After all, one usually learns the semantics of new programming languages before learning, if one ever does, the implementation of such languages. Later on it will be shown how a nondeterministic program can be translated into a deterministic program that computes a related function (see Section 4.3).

Nondeterministic instructions are essentially instructions that can choose between some given options. Although one is often required to make choices in everyday life, the use of such instructions might seem strange within the context of programs.

The semantics of a nondeterministic looping instruction of the form **do** α_1 **or** α_2 **or**... **or** α_k **until** $Q(x_1,...,x_m)$, are similar to those of a deterministic looping instruction of the form **do** α **until** $Q(x_1,...,x_m)$. The only difference is that in the deterministic case a fixed code segment α is executed in each iteration, whereas in the nondeterministic case an arbitrary code segment from $\alpha_1,...,\alpha_k$

is executed in each iteration. The choice of a code segment can differ from one iteration to another.

Example 1.3.6 The program in Figure 1.3.4 is nondeterministic. The set of natural numbers is assumed to be the domain of the variables, with 0 as initial value. Parenthetical remarks are enclosed between /* and */.

```
counter := 0
/* Choose five input values. */
do
    read value
or
    read value
    write value
    counter := counter + 1
until counter = 5
/* Read the remainder of the input .*/
do
    if eof then accept
    read value
until false
```

Fig 1.3.4 A nondeterministic program that chooses five input values

The program on input "1,2,3,4,5,6" has an execution sequence of the following form. The execution sequence starts with an iteration of the nondeterministic looping instruction in which the first code segment is chosen. The execution of the code segment consists of reading the input value 1, while writing nothing and leaving counter with the value of 0. Then the execution sequence continues with five additional iterations of the nondeterministic looping instruction. In each of the additional iterations, the second code segment is chosen. Each execution of the second code segment reads an input value, outputs the value that has been read, and increases the value of counter by 1. When counter reaches the value of 5, the execution sequence exits the first looping instruction. During the first iteration of the second looping instruction, the execution sequence halts due to the execution of the conditional accept instruction. The execution sequence is an accepting computation with output "2,3,4,5,6".

The program on input "1,2,3,4,5,6" has four additional execution sequences similar to the one above. The only difference is that the additional execution sequences, instead of ignoring the input value 1, ignore the input values 2, 3, 4, and 5, respectively. An execution sequence ignores an input value i by choosing to read the value in the first code segment of the nondeterministic looping instruction. The additional execution sequences are accepting computations with outputs "1,3,4,5,6", "1,2,4,5,6", "1,2,3,5,6", and "1,2,3,4,6", respectively.

The program on input "1,2,3,4,5,6" also has an accepting computation of the

following form. The computation starts with five iterations of the first looping instruction. In each of these iterations the second code segment of the nondeterministic looping instruction is executed. During each iteration an input value is read, that value is written into the output, and the value of counter is increased by 1. After five iterations of the nondeterministic looping instruction, counter reaches the value of 5, and the computation transfers to the deterministic looping instruction. The computation reads the input value 6 during the first iteration of the deterministic looping instruction, and terminates during the second iteration. The output of the computation is "1,2,3,4,5".

The program has $2^7 - 14$ execution sequences on input "1,2,3,4,5,6" that are not computations. $2^6 - 7$ of these execution sequences terminate due to trying to read beyond the input end by the first read instruction, and $2^6 - 7$ of these execution sequences terminate due to trying to read beyond the input end by the second read instruction. In each of these execution sequences at least two input values are ignored by consuming the values in the first code segment of the nondeterministic looping instruction. The execution sequences differ in the input values they choose to ignore.

None of the execution sequences of the program on input "1,2,3,4,5,6" is a nonaccepting computation, because the program has an accepting computation on such an input.

The program does not accept the input "1,2,3,4". On such an input the program has 2^5 execution sequences all of which are nonaccepting computations.

The first nondeterministic looping instruction of the program is used for choosing the output values from the inputs. Upon choosing five values the execution sequences continue to consume the rest of the inputs in the second deterministic looping instruction.

On inputs with fewer than five values the execution sequences terminate in the first nondeterministic looping instruction, upon trying to read beyond the end of the inputs.

The variable counter records the number of values chosen at steps during each execution sequence. □

A deterministic program has exactly one execution sequence on each input, and each execution sequence of a deterministic program is a computation. On the other hand, the last example shows that a nondeterministic program might have more than one execution sequence on a given input, and that some of the execution sequences might not be computations of the program.

Nondeterministic looping instructions have been introduced to allow selections between code segments. The motivation for introducing nondeterministic as-

signment instructions is to allow selections between values. Specifically, a nondeterministic assignment instruction of the form $x:=?$ assigns to the variable x an arbitrary value from the domain of the variables. The choice of the assigned value can differ from one encounter of the instruction to another.

```
/* Nondeterministically find a value that
a. appears exactly once in the input, and
b. is the last value in the input. */
last:=?
write last
/* Read the input values, until a value equal
to the one stored in last is reached. */
do
    read value
until value=last
/* Check for end of input. */
if eof then accept
reject
```

Fig 1.3.5 A nondeterministic program for determining a single appearance of the input value

Example 1.3.7 The program in Figure 1.3.5 is nondeterministic. The set of natural numbers is assumed to be the domain of the variables. The initial value is assumed to be 0.

The program accepts a given input if and only if the last value in the input does not appear elsewhere in the input. Such a value is also the output of an accepting computation. For instance, on input "1,2,3" the program has the output "3". On the other hand, on input "1,2,1" no output is defined since the program does not accept the input.

On each input the program has infinitely many execution sequences. Each execution sequence corresponds to an assignment of a different value to *last* from the domain of the variables.

An assignment to *last* of a value that appears in the input, causes an execution sequence to exit the looping instruction upon reaching such a value in the input. With such an assignment, one of the following cases holds.

a. The execution sequence is an accepting computation if the value assigned to *last* appears only at the end of the input (e.g, an assignment of 3 to *last* on input "1,2,3").

b. The execution sequence is a nonaccepting computation if the value at the end of the input appears more than once in the input (e.g, an assignment of 1 or 2 to *last* on input "1,2,1").

c. The execution sequence is not a computation if neither (a) nor (b) hold (e.g, an assignment of 1 or 2 to *last* on input "1,2,3").

An assignment to *last* of a value that does not appear in the input causes an execution sequence to terminate within the looping instruction upon trying to read beyond the end of the input. With such an assignment, one of the following cases hold.

a. The execution sequence is a nonaccepting computation if the value at the end of the input appears more than once in the input (e.g, an assignment to *last* of any natural number that differs from 1 and 2 on input "1,2,1").

b. The execution sequence is a nonaccepting computation if the input is empty (e.g, an assignment of any natural number to *last* on input " ").

c. The execution sequence is not a computation, if neither (a) nor (b) hold (e.g, an assignment to *last* of any natural number that differs from 1,2, and 3 on input "1,2,3"). □

Intuitively, each program on each input defines "good" execution sequences, and "bad" execution sequences. The good execution sequences terminate due to the accept commands, and the bad execution sequences do not terminate due to accept commands. The best execution sequences for a given input are the computations that the program has on the input. If there exist good execution sequences, then the set of computations is identified with that set. Otherwise, the set of computations is identified with the set of bad execution sequences.

The computations of a program on a given input are either all accepting computations or all nonaccepting computations. Moreover, some of the nonaccepting computations may never halt. On inputs that are accepted the program might have execution sequences that are not computations. On the other hand, on inputs that are not accepted all the execution sequences are computations.

Guessing in Programs

The semantics of each program are characterized by the computations of the program. In the case of deterministic programs the semantics of a given program are directly related to the semantics of its instructions. That is, each execution of the instructions keeps the program within the course of a computation.

In the case of nondeterministic programs a distinction is made between execution sequences and computations, and so the semantics of a given program are related only in a restricted manner to the semantics of its instructions. That is, although each computation of the program can be achieved by executing the instructions, some of the execution sequences do not correspond to any computation of the program. The source for this phenomenon is the ability of

the nondeterministic instructions to make arbitrary choices.

Each program can be viewed as having an imaginary agent with magical power that executes the program. On a given input, the task of the imaginary agent is to follow any of the computations the program has on the input. The case of deterministic programs can be considered as a lesser and restricted example in which the agent is left with no freedom. That is, the outcome of the execution of each deterministic instruction is completely determined for the agent by the semantics of the instruction. On the other hand, when executing a nondeterministic instruction the agent must satisfy not only the local semantics of the instruction, but also the global goal of reaching an accept command whenever the global goal is achievable.

Specifically, the local semantics of a nondeterministic looping instruction of the form **do** α_1 **or**... **or** α_k **until** $Q(x_1,...,x_m)$ require that in each iteration exactly one of the code segments $\alpha_1,...,\alpha_k$ will be chosen in an arbitrary fashion by the agent. The global semantics of a program require that the choice be made for a code segment which can lead the execution sequence to halt due to a conditional accept instruction, whenever such is possible.

Similarly, the local semantics of a nondeterministic assignment instruction of the form $x:=?$ require that each assigned value of x be chosen by the agent in an arbitrary fashion from the domain of the variables. The global semantics of the program require that the choice be made for a value that halts the execution sequence due to a conditional accept instruction, whenever such is possible.

From the discussion above it follows that the approach of "first guess a solution and then check for its validity" can be used when writing a program. This approach simplifies the task of the programmer whenever checking for the validity of a solution is simpler than the derivation of the solution. In such a case, the burden of determining a correct "guess" is forced on the agent performing the computations.

It should be emphasized that from the point of view of the agent, a guess is correct if and only if it leads an execution sequence along a computation of the program. The agent knows nothing about the problem that the program intends to solve. The only thing that drives the agent is the objective of reaching the execution of a conditional accept instruction at the end of the input. Consequently, it is still up to the programmer to fully specify the constraints that must be satisfied by the correct guesses.

Example 1.3.8 The program of Figure 1.3.6 outputs a value that does not appear in the input. The program starts each computation by guessing a value and storing it in x. Then the program reads the input and checks that each of the input values differs from the value stored in x. □


```
/* Guess the output value. */
x :=?
write x
/* Check for the correctness of the guessed value. */
do
    if eof then accept
    read y
until y = x
```

Fig 1.3.6 A nondeterministic program that outputs a noninput value

The notion of an imaginary agent provides an appealing approach for explaining nondeterminism. Nevertheless, the notion should be used with caution to avoid misconceptions. In particular, an imaginary agent should be employed only on full programs. The definitions leave no room for one imaginary agent to be employed by other agents. For instance, an imaginary agent that is given the program P in the following example cannot be employed by other agents to derive the acceptance of exactly those inputs that the agent rejects.

Example 1.3.9 Consider the program P in Figure 1.3.7. On a given input, P outputs an arbitrary choice of input values, whose sum equals the sum of the nonchosen input values. The values have the same relative ordering in the output as in the input.

```
sum1 := 0
sum2 := 0
do
    if eof then accept
    do /* Guess where the next input value belongs. */
        read x
        sum1 := sum1 + x
    or
        read x
        write x
        sum2 := sum2 + x
    until sum1 = sum2
    /* Check for the correctness of the guesses, with
    respect to the portion of the input consumed so far. */
until false
```

Fig 1.3.7 A nondeterministic program for partitioning the input into two subsets of equal sums of elements

For instance, on input "2,1,3,4,2" the possible outputs are "2,1,3", "1,3,2", "2,4", and "4,2". On the other hand, no output is defined for input "2,3".

In each iteration of the nested looping instruction the program guesses whether

the next input value is to be among the chosen ones. If it is to be chosen then *sum*2 is increased by the magnitude of the input value. Otherwise, *sum*1 is increased by the magnitude of the input value. The program checks that the sum of the nonchosen input values equals the sum of the chosen input values by comparing the value in *sum*1 with the value in *sum*2. □

Example 1.3.10 The program of Figure 1.3.8 outputs the median of its input values, that is, the $[n/2]$th smallest input value for the case that the input consists of n values. On input "1,3,2" the program has the output "2", and on input "2,1,3,3" the program has the output "3".

The program starts each computation by storing in median a guess for the value of the median. Then the program reads the input values and determines in count the difference between the number of input values that are greater than the one stored in median and the number of input values that are smaller than the one stored in median.

For those input values that are equal to the value stored in median, the program guesses whether they should be considered as bigger values or smaller values.

The program checks that the guesses are correct by verifying that count holds either the value 0 or the value 1. □

```
median:=?      /* Guess the median. */
write median
count:=0
do
    /* Find the difference between the number of values greater
    than and those smaller than the guessed median. */
    do
        read x
        if x>median then
            count:=count+1
        if x<median then
            count:=count-1
        if x=median then
            do
                count:=count+1
            or
                count:=count-1
            until true
    until 0≤count≤1
    /* The median is correct for the portion of the input consumed so far. */
    if eof then accept
until false
```

Fig 1.3.8 A nondeterministic program that finds the median of the input values

The *relation computed* by a program P, denoted $R(P)$, is the set

$$\{(x,y) \mid P \text{ has an accepting computation on input } x \text{ with output } y\}.$$

When P is a deterministic program, the relation $R(P)$ is a function.

Example 1.3.11 Consider the program P in Figure 1.3.6. Assume the set of natural numbers for the domain of the variables. The relation $R(P)$ that P computes is

$$\{(\alpha,a) \mid \alpha \text{ is a sequence of natural numbers, and } a \text{ is a natural number that does not appear in } \alpha\}.$$ □

The language that a program P accepts is denoted by $L(P)$ and it consists of all the inputs that P accepts.

Configurations of Programs

An execution of a program on a given input is a discrete process in which the input is consumed, an output is generated, the variables change their values, and the program traverses its instructions. Each stage in the process depends on the outcome of the previous stage, but not on the history of the stages. The outcome of each stage is a configuration of the program that indicates the instruction being reached, the values stored in the variables, the portion of the input left to be read, and the output that has been generated so far. Consequently, the process can be described by a sequence of moves between configurations of the program.

Formally, a segment of a program is said to be an *instruction segment* if it is of any of the following forms.

a. Read instruction
b. Write instruction
c. Assignment instruction
d. **if** $Q(x_1,...,x_m)$ **then** portion of a conditional if instruction
e. **do** portion of a looping instruction
f. **until** $Q(x_1,...,x_m)$ portion of a looping instruction
g. Conditional accept instruction
h. Reject instruction

Consider a program P that has k instruction segments, m variables, and a domain of variables that is denoted by D. A *configuration*, or *instantaneous description*, of P is a five-tuple (i,x,u,v,w), where $1 \leq i \leq k$, x is a sequence of m values from D, and u, v, and w are sequences of values from D.

Intuitively, a configuration (i,x,u,v,w) says that P is in its ith instruction segment, its jth variable contains the jth value of x, u is the portion of the input that has already been read, the leftover of the input is v, and the output so far is w. (The component u is not needed in the definition of a configuration. It is inserted here for reasons of compatibility with future definitions that require such a component.)

```
last:=?                  /* I1 */
write last               /* I2 */
do                       /* I3 */
    read value           /* I4 */
until value=last         /* I5 */
if eof then accept       /* I6 */
reject                   /* I7 */
```

Fig. 1.3.9 A program consisting of seven instruction segments.

Example 1.3.12 Consider the program in Figure 1.3.9. Assume the set of natural numbers for the domain D of the variables, with 0 as initial value. Each line I_i in the program is an instruction segment. The program has $k=7$ instruction segments, and $m=2$ variables.

In each configuration (i,x,u,v,w) of the program i is a natural number between 1 and 7, and x is a pair $\langle last,value\rangle$ of natural numbers that corresponds to a possible assignment of last and value in the variables *last* and *value*, respectively. Similarly, u, v, and w are sequences of natural numbers.

The configuration $(1,\langle 0,0\rangle,\langle\rangle,\langle 1,2,3\rangle,\langle\rangle)$ states that the program is in the first instruction segment, the variables hold the value 0, no input value has been read so far, the rest of the input is "1,2,3", and the output is empty.

The configuration $(5,\langle 3,2\rangle,\langle 1,2\rangle,\langle 3\rangle,\langle 3\rangle)$ states that the program is in the fifth instruction segment, the variable *last* holds the value 3, the variable *value* holds the value 2, "1,2" is the portion of the input consumed so far, the rest of the input contains just the value 3, and the output so far contains only the value 3. □

A configuration (i,x,u,v,w) of P is called an *initial configuration* if $i=1$, x is a sequence of m initial values, u is an empty sequence, and w is an empty sequence. The configuration is said to be an *accepting configuration* if the ith instruction segment of P is a conditional accept instruction and v is an empty sequence.

A direct move of P from configuration C_1 to configuration C_2 is denoted $C_1 \vdash_P C_2$, or simply $C_1 \vdash C_2$ if P is understood. A sequence of unspecified number of moves of P from configuration C_1 to configuration C_2 is denoted $C_1 \vdash^*_P C_2$, or simply $C_1 \vdash^* C_2$ if P is understood.

Example 1.3.13 Consider the program in Figure 1.3.9. On input "1,2,3" it has an accepting computation that goes through the following sequence of moves between configurations. The first configuration in the sequence is the initial configuration of the program on input "1,2,3", and the last configuration in the sequence is an accepting configuration of the program. In each configuration (i,x,u,v,w) the pair $x=\langle last,value\rangle$ corresponds to the assignment of `last` and `value` in the variables **last** and *value*, respectively.

$$
\begin{aligned}
(1,\langle 0,0\rangle,\langle\rangle,\langle 1,2,3\rangle,\langle\rangle) &\vdash (2,\langle 3,0\rangle,\langle\rangle,\langle 1,2,3\rangle,\langle\rangle)\\
&\vdash (3,\langle 3,0\rangle,\langle\rangle,\langle 1,2,3\rangle,\langle 3\rangle)\\
&\vdash (4,\langle 3,0\rangle,\langle\rangle,\langle 1,2,3\rangle,\langle 3\rangle)\\
&\vdash (5,\langle 3,1\rangle,\langle 1\rangle,\langle 2,3\rangle,\langle 3\rangle)\\
&\vdash (3,\langle 3,1\rangle,\langle 1\rangle,\langle 2,3\rangle,\langle 3\rangle)\\
&\vdash (4,\langle 3,1\rangle,\langle 1\rangle,\langle 2,3\rangle,\langle 3\rangle)\\
&\vdash (5,\langle 3,2\rangle,\langle 1,2\rangle,\langle 3\rangle,\langle 3\rangle)\\
&\vdash (3,\langle 3,2\rangle,\langle 1,2\rangle,\langle 3\rangle,\langle 3\rangle)\\
&\vdash (4,\langle 3,2\rangle,\langle 1,2\rangle,\langle 3\rangle,\langle 3\rangle)\\
&\vdash (5,\langle 3,3\rangle,\langle 1,2,3\rangle,\langle\rangle,\langle 3\rangle)\\
&\vdash (6,\langle 3,3\rangle,\langle 1,2,3\rangle,\langle\rangle,\langle 3\rangle)
\end{aligned}
$$

The subcomputation

$$(1,\langle 0,0\rangle,\langle\rangle,\langle 1,2,3\rangle,\langle\rangle) \vdash^* (1,\langle 0,0\rangle,\langle\rangle,\langle 1,2,3\rangle,\langle\rangle)$$

consists of zero moves, and the subcomputation

$$(1,\langle 0,0\rangle,\langle\rangle,\langle 1,2,3\rangle,\langle\rangle) \vdash^* (6,\langle 3,3\rangle,\langle 1,2,3\rangle,\langle\rangle,\langle 3\rangle)$$

consists of eleven moves. □

1.4 Problems

The first two sections of this chapter treated different aspects of information. The third section considered programs. The purpose of the rest of this chapter is to deal with the motivation for writing programs for manipulating information, that is, with problems.

Each *problem* K is a pair consisting of a set and a question, where the question can be applied to each element in the set. The set is called the *domain* of the problem, and its elements are called the *instances* of the problem.

Example 1.4.1 Consider the problem K_1 defined by the following domain and question.

Domain: $\{\langle a,b\rangle \mid a \text{ and } b \text{ are natural numbers}\}$.

Question: What is the integer part y of a divided by b for the given instance $x=\langle a,b\rangle$? The domain of the problem contains the instances $\langle 0,0\rangle$, $\langle 5,0\rangle$, $\langle 3,8\rangle$, $\langle 24,6\rangle$, and $\langle 27,8\rangle$. On the other hand, $\langle -5,3\rangle$ is not an instance of the problem.

For the instance $\langle 27,8\rangle$ the problem asks what is the integer part of 27 divided by 8. Similarly, for the instance $\langle 0,0\rangle$ the problem asks what is the integer part of 0 divided by 0. □

An answer to the question that the problem K poses for a given instance is said to be a *solution* for the problem at the given instance. The *relation induced by the problem*, denoted $R(K)$, is the set

$$\{(x,y) \mid x \text{ is an instance of the problem, and } y \text{ is a solution for the problem at } x\}.$$

The problem is said to be a *decision problem* if for each instance the problem has either a yes or a no solution.

Example 1.4.2 Consider the problem K_1 in Example 1.4.1. The problem has the solution 3 at instance $\langle 27,8\rangle$, and an undefined solution at instance $\langle 0,0\rangle$. K_1 induces the relation

$$R(K_1)=\{(\langle 0,1\rangle,0),(\langle 0,2\rangle,0),(\langle 1,1\rangle,1),(\langle 0,3\rangle,0),(\langle 1,2\rangle,0),(\langle 2,1\rangle,2),(\langle 0,3\rangle,0),\ldots\}.$$

The problem K_1 is not a decision problem. But the problem K_2 defined by the following pair is.

Domain: $\{\langle a,b\rangle \mid a \text{ and } b \text{ are natural numbers}\}$.

Question: Does b divide a, for the given instance $\langle a,b\rangle$? □

Partial Solvability and Solvability

A program P is said to *partially solve* a given problem K if it provides the answer for each instance of the problem, that is, if $R(P)=R(K)$. If, in addition, all the computations of the program are halting computations, then the program is said to *solve* the problem.

Example 1.4.3 Consider the program P_1 in Figure 1.4.1(a). The domain of the variables is assumed to equal the set of natural numbers. The program partially solves the problem K_1 of Example 1.4.1.

(a)
```
read a
read b
ans:=0
if a ⩾ b then
    do
        a:=a−b
        ans:=ans+1
    until a<b
write ans
if eof then accept
```

(b)
```
read a
read b
do
    if a=b then
        if eof then accept
    a:=a−b
until false
```

Fig. 1.4.1

(a) A program that partially solves the problem of dividing natural numbers
(b) A program that partially decides the problem of divisibility of natural numbers

On input "27,8" the program halts in an accepting configuration with the answer 3 in the output. On input "0,0" the program never halts, and so the program has undefined output on such an input. On input "27" and input "27,8,2" the program halts in rejecting configurations and does not define an output.

The program P_1 does not solve K_1 because it does not halt when the input value for b is 0. P_1 can be modified to a program P that solves K_1 by letting P check for a 0 assignment to b. □

A program is said to *partially decide* a problem if the following two conditions are satisfied.

a. The problem is a decision problem; and

b. The program accepts a given input if and only if the problem has an answer yes for the input, that is, the program accepts the language

$$\{x \mid x \text{ is an instance of the problem, and the problem has the answer yes at } x\}.$$

A program is said to *decide* a problem if it partially decides the problem and all its computations are halting computations.

Example 1.4.4 It is meaningless to talk about the partial decidability or decidability of the problem K_1 of Example 1.4.1 by a program, because the problem is not a decision problem. On the other hand, the problem K_2 of Example 1.4.2 is a decision problem. The latter problem is partially decidable by the program P_2 in Figure 1.4.1(b). □

The main difference between a program P_1 that partially solves (partially decides) a problem, and a program P_2 that solves (decides) the same problem,

is that P_1 might reject an input by a nonhalting computation, whereas P_2 can reject the input only by a halting computation. (Recall that on an input that is accepted by a program, the program has only accepting computations, and all these computations are halting computations. But on an input that is not accepted the program might have more than one computation, of which some may never halt.)

The notions of partial solvability, solvability, partial decidability, and decidability of a problem by a program can be intuitively generalized in a straightforward manner by considering effective (i.e., strictly mechanical) procedures instead of programs. However, formalizing the generalizations requires that the intuitive notion of effective procedure be formalized. In any case, under such intuitively understood generalizations a problem is said to be *partially solvable*, *solvable*, *partially decidable*, and *decidable* if it can be partially solved, solved, partially decided, and decided by an effective procedure, respectively.

In what follows effective procedures will also be called *algorithms*.

Example 1.4.5 The program P_1 of Example 1.4.3 describes how the problem K_1 of Example 1.4.1 can be solved. The program P_2 of Example 1.4.4 describes how the problem K_2 of Example 1.4.2 can be solved. □

A problem is said to be *unsolvable* if no algorithm can solve it. The problem is said to be *undecidable* if it is a decision problem and no algorithm can decide it. The relationship between the different classes of problems is illustrated in the Venn diagram of Figure 1.4.2.

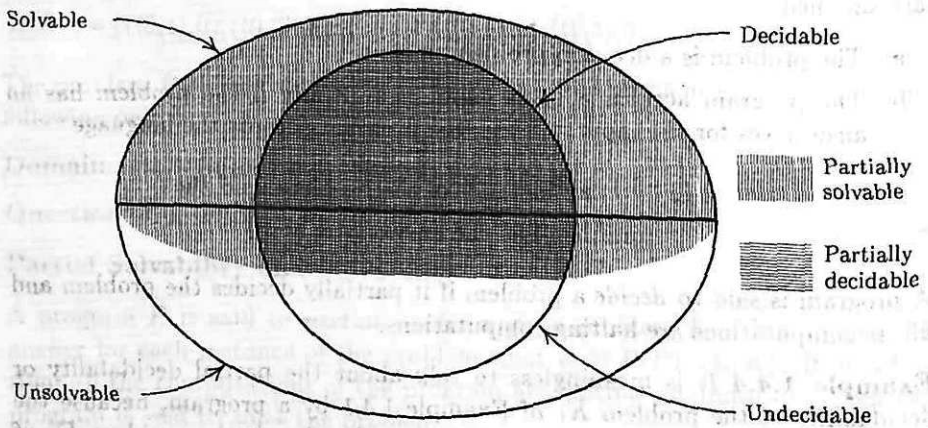


Fig. 1.4.2. Classification of the set of problems

It should be noted that an unsolvable problem might be partially solvable by an algorithm that makes an exhaustive search for a solution. In such a case the solution is eventually found whenever it is defined, but the search might

continue forever whenever the solution is undefined. Similarly, an undecidable problem might also be partially decidable by an algorithm that makes an exhaustive search. However, here the solution is eventually found whenever it has the value **yes**, but the search might continue forever whenever value **no**.

Example 1.4.6 The empty-word membership problem for grammars is the problem consisting of the following domain and question.

Domain: $\{G \mid G \text{ is a grammar}\}$.

Question: Is the empty word ε in $L(G)$ for the given grammar G ? It is possible to show that the problem is undecidable (e.g, see Theorem 4.6.2 and Exercise 4.5.7). On the other hand, the problem is partially decidable because given an instance $G=\langle N,\Sigma,P,S\rangle$ one can exhaustively search for a derivation of the form $S=\gamma_0 \Rightarrow \gamma_1 \Rightarrow \ldots \Rightarrow \gamma_{n-1} \Rightarrow \gamma_n = \varepsilon$, by considering all derivations of length n for $n=1,2,\ldots$ With such an algorithm the desired derivation will eventually be found if ε is in $L(G)$. However, if ε is not in $L(G)$, then the search might never terminate.

For the grammar $G=\langle N,\Sigma,P,S\rangle$, whose production rules are listed below, the algorithm will proceed in the following manner.

$$\begin{aligned} S &\rightarrow aBS \\ &\rightarrow Ba \\ aB &\rightarrow SB \\ BS &\rightarrow \varepsilon \end{aligned}$$

The algorithm will start by determining the set of all the derivations $\Psi_1=\{S \Rightarrow aBS,\ S \Rightarrow Ba\}$ of length $n=1$. After determining that none of the derivations in Ψ_1 provides the empty string ε, the algorithm determines the set of all the derivations $\Psi_2=\{S \Rightarrow aBS \Rightarrow aBaBS,\ S \Rightarrow aBS \Rightarrow aBBa,\ S \Rightarrow aBS \Rightarrow SBS,\ S \Rightarrow aBS,\ S \Rightarrow a\}$ of length $n=2$. Then the algorithm continues by determining the set Ψ_3 of all the derivations of length 3, the set Ψ_4 of all the derivations of length 4, and so forth. The algorithm stops (with the answer **yes**) when, and only when, it finds a set Ψ_n of derivations of length n that includes a derivation for ε. Such a set Ψ_n exists for $n=5$ because of the derivation $S \Rightarrow aBS \Rightarrow SBS \Rightarrow BaBS \Rightarrow BSBS \Rightarrow BS \Rightarrow \varepsilon$.

On the other hand, for the grammar $G=\langle N,\Sigma,P,S\rangle$, whose production rules are listed below, the algorithm never stops.

$$\begin{aligned} S &\rightarrow aSb \\ &\rightarrow aAb \\ A &\rightarrow \varepsilon \end{aligned}$$

□

The unsolvability of a problem does not mean that a solution cannot be found at some of its instances. It just means that no algorithm can uniformly find a solution for every given instance. Consequently, an unsolvable problem might have simplified versions that are solvable. The simplifications can be in the question being asked and in the domain being considered.

Example 1.4.7 The empty-word membership problem for Type 1 grammars is the problem consisting of the following domain and question.

Domain: $\{G \mid G$ is a Type 1 grammar $\}$.

Question: Is the empty word ε in $L(G)$ for the given grammar G ? The problem is decidable because ε is in $L(G)$ for a given Type 1 grammar $G = \langle N,\Sigma,P,S\rangle$ if and only if $S \to \varepsilon$ is a production rule of G. □

A function f is said to be *computable* (respectively, *partially computable*, *non-computable*) if the problem defined by the following domain and question is solvable (respectively, partially solvable, unsolvable).

Domain: The domain of f.

Question: What is the value of $f(x)$ at the given instance x ?

Problems concerning Programs

Although programs are written to solve problems, there are also interesting problems that are concerned with programs. The following are some examples of such decision problems.

Uniform halting problem for programs

Domain: Set of all programs.

Question: Does the given program halt on each of its inputs, that is, are all the computations of the program halting computations ?

Halting problem for programs

Domain: $\{(P,x) \mid P$ is a program and x is an input for $P\}$.

Question: For the given instance (P,x) does P halt on x, that is, are all the computations of P on input x halting computations ?

Recognition / acceptance problem for programs

Domain: $\{(P,x) \mid P$ is a program and x is an input for $P\}$.

Question: For the given instance (P,x) does P accept x ?

Membership problem for programs

Domain: $\{(P,x,y) \mid P$ is a program, and x and y are sequences of values from the domain of the variables of $P\}$.

Question: Is (x,y) in the relation $R(P)$ for the given instance (P,x,y), that is, does P have an accepting computation on x with output y ?

Emptiness problem for programs

Domain: Set of all programs.

Question: Does the given program accept an empty language, that is, does the program accept no input ?

Ambiguity problem for programs

Domain: Set of all programs.

Question: Does the given program have two or more accepting computations that define the same output for some input ?

Single-valuedness problem for programs

Domain: Set of all programs.

Question: Does the given program define a function, that is, does the given program for each input have at most one output ?

Equivalence problem for programs

Domain: $\{(P_1,P_2) \mid P_1$ and P_2 are programs $\}$.

Question: Does the given pair (P_1,P_2) of programs define the same relation, that is, does $R(P_1) = R(P_2)$?

Example 1.4.8 The two programs P_1 and P of Example 1.4.3 are equivalent, but only P_2 halts on all inputs. □

The nonuniform halting problem, the unambiguity problem, the inequivalence problem, and so forth are defined similarly for programs as the uniform halting problem, the ambiguity problem, the equivalence problem, and so forth, respectively. The only difference is that the questions are phrased so that the solutions to the new problems are the complementation of the old ones, that is, **yes** becomes **no** and **no** becomes **yes**.

It turns out that nontrivial questions about programs are difficult, if not impossible, to answer. It is natural to expect these problems to become easier when the programs under consideration are "appropriately" restricted. The extent to which the programs have to be restricted, as well as the loss in their expressibility power, and the increase in the resources they require due to such restrictions, are interesting questions on their own.

Problems concerning Grammars

Some of the problems concerned with programs can in a similar way be defined also for grammars. The following are some examples of such problems.

Membership problem for grammars

Domain: $\{(G,x) \mid G$ is a grammar $\langle N,\Sigma,P,S\rangle$ and x is a string in $\Sigma^*\}$.

Question: Is x in $L(G)$ for the given instance (G,x) ?

Emptiness problem for grammars

Domain: $\{G \mid G$ is a grammar $\}$.

Question: Does the given grammar define an empty language ?

Ambiguity problem for grammars

Domain: $\{G \mid G$ is a grammar$\}$.

Question: Does the given grammar G have two or more different derivation graphs for some string in $L(G)$?

Equivalence problem for grammars

Domain: $\{(G_1,G_2) \mid G_1$ and G_2 are grammars $\}$.

Question: Does the given pair of grammars generate the same language ?

1.5 Reducibility among Problems

A common approach in solving problems is to transform them to different problems, solve the new ones, and derive the solutions for the original problems from those for the new ones. This approach is useful when the new problems are simpler to solve, or when they already have known algorithms for solving them. A similar approach is also very useful in the classification of problems according to their complexity.

A problem K_1, which can be transformed to another problem K_2, is said to be reducible to the new problem. Specifically, a problem K_1 is said to be *reducible* to a problem K_2 if there exist computable total functions f and g with the following properties (see Figure 1.5.1).

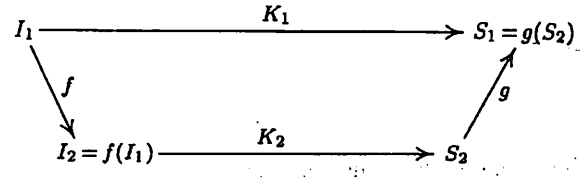


Fig. 1.5.1. Reducibility from problem K_1 to problem K_2

If I_1 is an instance of K_1, then

a. $I_2 = f(I_1)$ is an instance of K_2.

b. K_1 has a solution S_1 at I_1 if and only if K_2 has a solution S_2 at $I_2 = f(I_1)$ such that $S_1 = g(S_2)$.

Example 1.5.1 Let Ψ be the set $\{m \mid m = 2^i$ for some integer $i\}$. The problem of exponentiation of numbers from Ψ is reducible to the problem of multiplication of integer numbers. The reducibility is implied from the equalities $x^y = (2^{\log x})^y = 2^{y \cdot \log x}$, which allow the choice of $f(x,y) = (y, \log x)$ and $g(z) = 2^z$ for f and g, respectively. □

Example 1.5.2 Let $K_\varnothing$ and $K_\equiv$ be the emptiness problem and the equivalence problem for programs, respectively. Then $K_\varnothing$ is reducible to $K_\equiv$ by functions f and g of the following form. f is a function whose value at program P is the pair of programs $(P,P_\varnothing)$. $P_\varnothing$ is a program that accepts no input, for example, the program that consists only of the instruction **reject**. g is the identity function, that is, $g(\texttt{yes}) = \texttt{yes}$ and $g(\texttt{no}) = \texttt{no}$. □

If K_1 is a problem that is reducible to a problem K_2 by total functions f and g that are computable by algorithms T_f and T_g, respectively, and K_2 is solvable by an algorithm A, then one can also get an algorithm B for solving K_1 (see Figure 1.5.2).

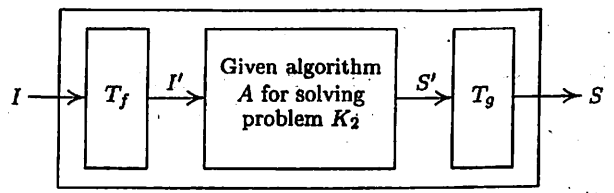


Fig. 1.5.2. Reduction of problem K_1 to problem K_2

Given an input I, the algorithm B starts by running T_f on I. Then B gives the output I' of T_f to A. Finally B gives the output S' of A to T_g, and assumes the same output S as the one obtained from T_g.

1.6 Exercises

1.1.1

a. Find all the alphabets that use only symbols from the set $\{a,b,c\}$. Which of the alphabets is unary ? Which is binary ?

b. Let S be a set of t symbols. How many unary alphabets can be constructed from the symbols of S ? How many binary alphabets ?

1.1.2 For each of the following conditions find all the strings α over the alphabet $\{a,b\}$ that satisfy the condition.

a. No symbol is repeated in α.

b. The length of α is 3, that is, $|\alpha|=3$.

1.1.3

a. Find $\alpha\beta,\beta\alpha,\alpha^2,\alpha^0\beta^2$, and $\alpha^2\beta^2\varepsilon$ for the case that $\alpha=a$ and $\beta=ab$.

b. Find all the pairs of strings α and β over the alphabet $\{a,b\}$ that satisfy $\alpha\beta=abb$.

1.1.4 Let α be the string 011.

a. Find all the proper prefixes of α^2.

b. Find all the substrings β of $\alpha\alpha^{rev}$ that satisfy $\beta=\beta^{rev}$.

1.1.5 How many strings of length t are in Σ^* if Σ is an alphabet of cardinality r.

1.1.6 For each of the following cases give the first 20 strings in $\{a,b,c\}^*$.

a. $\{a,b,c\}^*$ is given in alphabetical ordering.

b. $\{a,b,c\}^*$ is given in canonical ordering.

1.1.7 Let S be the set of all the strings over the alphabet $\{a,b,c\}$, that is, $S=\{a,b,c\}^*$. Let S_1 and S_2 be subsets of S. Which are the strings that appear both in S_1 and in S_2 in each of the following cases ?

a. S_1 contains the t alphabetically smallest strings in S, and S_2 contains all the strings in S of length t at most.

b. S_1 contains the 127 alphabetically smallest strings in S, and S_2 contains the 127 canonically smallest strings in S.

1.1.8 Show that if Σ is an alphabet, then Σ^* has the following representations.

a. Binary representation

b. Unary representation

1.1.9 Find a binary representation for the set of rational numbers.

1.1.10 Show that if D has a binary representation f_1, then it also has a binary representation f_2, such that $f_2(e)$ is an infinite set for each element e of D.

1.1.11 Let f_1 and f_2 be binary representations for D_1 and D_2, respectively. Find a binary representation f for each of the following sets.

a. $D_1 \cup D_2$

b. $D_1 \times D_2$

c. D_1^*

1.1.12 Show that the set of real numbers does not have a binary representation.

1.2.1 Let L be the language $\{\varepsilon,0,10\}$. Determine the following sets.

a. $L\cup\overline{L}$

b. $L\cap\overline{L}$

c. $L\overline{L}$

d. $\overline{L}L$

e. L^2

f. $L\times L$

1.2.2 Let $G=\langle N,\Sigma,P,S\rangle$ be a grammar in which $N=\{S\}$, $\Sigma=\{a\}$, and each production rule contains at most 3 symbols. What are the possible production rules in P ?

1.2.3 Let G be the grammar $\langle N,\Sigma,P,S\rangle$, where $N=\{S\}$, $\Sigma=\{a,b\}$, and $P=\{S\rightarrow\varepsilon,\ S\rightarrow aSbS\}$.

a. Find all the strings that are directly derivable from SaS in G.

b. Find all the derivations in G that start at S and end at ab.

c. Find all the sentential forms of G of length 4 at most.

1.2.4 Find all the derivations of length 3 at most that start at S in the grammar $\langle N,\Sigma,P,S\rangle$ whose production rules are listed below.

$$S \rightarrow AS$$
$$aS \rightarrow bb$$
$$A \rightarrow aa$$

1.2.5 For each of the following sets of production rules P find all the strings of length 4 or less in the language generated by the grammar $\langle N,\Sigma,P,S\rangle$.

(a)
$$\begin{aligned} S &\to aa \\ &\to bb \\ &\to aSa \\ &\to bSb \\ &\to SS \end{aligned}$$

(b)
$$\begin{aligned} S &\to aSA \\ &\to bSB \\ &\to X \\ aA &\to Aa \\ bA &\to Ab \\ aB &\to Ba \\ bB &\to Bb \\ XA &\to Xa \\ &\to a \\ XB &\to Xb \\ &\to b \end{aligned}$$

1.2.6 Give two parse trees for the string $aababb$ in the grammar $G=\langle N,\Sigma,P,S\rangle$, whose production rules are listed below.

$$\begin{aligned} S &\to b \\ &\to aA \\ &\to aS \\ A &\to Ab \\ &\to Sa \\ &\to a \end{aligned}$$

1.2.7 Consider the grammar $G=\langle N,\Sigma,P,E\rangle$ for arithmetic expressions, where $N=\{E,T,F\}$, $\Sigma=\{+,*,(,),a\}$, and P is the set consisting of the following production rules.

$$\begin{aligned} E &\to E+T \\ E &\to T \\ T &\to T*F \\ &\to F \\ F &\to (E) \\ &\to a \end{aligned}$$

Give the derivation tree for the expression $a*(a+a)$ in G.

1.2.8 Let $G=\langle N,\Sigma,P,S\rangle$ be the grammar with the following production rules. Find the derivation graph for the string $a^3b^3c^3$ in G.

$$\begin{aligned} S &\to aSBc \\ &\to \varepsilon \\ cB &\to Bc \\ aB &\to ab \\ bB &\to bb \end{aligned}$$

1.2.9 For each of the following languages give a grammar that generates the language.

a. $\{x01 \mid x \text{ is in } \{0,1\}^*\}$

b. $\{01,10\}^*$

c. $\{x \mid x \text{ is in } \{a,b\}^*$, and the number of a's in x or the number of b's in x is equal to 1$\}$

d. $\{0^i1^j \mid i \geqslant j\}$

e. $\{x \mid x \text{ is in } \{0,1\}^*$, and $x=x^{rev}\}$

f. $\{x \mid x \text{ is in } \{0,1\}^*$, and each 01 in x is followed by 10$\}$

g. $\{x \mid x \text{ is in } \{0,1\}^*$, and the length of x is not divisible by 3$\}$

h. $\{x \mid x \text{ is in } \{a,b\}^*$, and x is of odd length if and only if it ends with $b\}$

i. $\{x \mid x \text{ is in } \{a,b\}^*$, and abb is not a substring of $x\}$

j. $\{x\#y \mid x \text{ is in } \{a,b\}^*$, and y is a permutation of $x\}$

k. $\{a^ib^ic^id^i \mid i \text{ is a natural number }\}$

l. $\{a^{i_1}\#a^{i_2}\#a^{i_3}\#\ldots\#a^{i_n} \mid n \geqslant 2 \text{ and } i_j=i_k \text{ for some } 1 \leqslant j < k \leqslant n\}$

m. $\{aba^2b^2a^3b^3\ldots a^nb^n \mid n \text{ is a natural number }\}$

1.2.10 For each of the following conditions show how, from any arbitrary given pair of grammars $G_1=\langle N_1,\Sigma_1,P_1,S_1\rangle$ and $G_2=\langle N_2,\Sigma_2,P_2,S_2\rangle$, a grammar $G_3=\langle N_3,\Sigma_3,P_3,S_3\rangle$ that satisfies the condition can be constructed.

a. $L(G_3)=\{w \mid w^{rev} \text{ is in } L(G_1)\}$

b. $L(G_3)=L(G_1)\cup L(G_2)$

c. $L(G_3)=L(G_1)L(G_2)$

d. $L(G_3)=(L(G_1))^*$

e. $L(G_3)=L(G_1)\cap L(G_2)$

1.2.11 Show that from each grammar $G_1=\langle N,\Sigma,P_1,S\rangle$ a two-nonterminal-symbols grammar $G_2=\langle\{S,A\},\Sigma,P_2,S\rangle$ can be constructed such that $L(G_1)=L(G_2)$.

1.2.12 Give the leftmost derivation and a nonleftmost derivation for the string $abbabb$ in the grammar of Exercise 1.2.5.

1.2.13 For each of the following cases find all the possible grammars G that satisfy the case, where $G=\langle N,\Sigma,P,S\rangle$ with $N=\{S\}$, $\Sigma=\{a,b\}$, and P being a subset of $\{S\to\varepsilon,\ S\to abAS,\ S\to ab,\ bA\to aS\}$.

a. G is a Type 3 grammar.

b. G is a Type 2 grammar, but not of Type 3.

c. G is a Type 1 grammar, but not of Type 2.

1.3.1 Consider the program P in Figure 1.E.1(a). What are the outputs of P on input "2,2" ? On input "3,2" ?

(a)
```
sum1:=0
sum2:=0
do
    if eof then accept
    do
        read x
        sum1:=sum1+x
    or
        read x
        write x
        sum2:=sum2+x
    until sum1≠sum2
until false
```

(b)
```
x:=?
write x
do
    if eof then accept
    read y
until x≠y
```

Figure 1.E.1

1.3.2 Consider the program P in Figure 1.E.1(b). Assume that P has the domain of variables $\{0,1,2,3,4,5\}$. What are the outputs of P on input "2,2" ? On input "3,2" ?

1.3.3 For each of the following cases write a program that corresponds to the case. Assume that the variables have the set of natural numbers as the domain of the variables, and 0 as an initial value..

a. The program outputs an input value v such that $v+2$ does not appear in the input.
Example : On input "1,4,2,3" the program should have an accepting computation with output "3", and an accepting computation with output "4". Moreover, each accepting computation of the program should provide either the output "3" or the output "4".

b. The program outputs an input value v such that $v+2$ also appears in the input.
Example : On input "1,4,3,2" the program should have an accepting

computation with output "1", and an accepting computation with output "2". Moreover, each accepting computation of the program should provide either the output "1" or the output "2".

c. The program outputs a value that does not appear exactly twice in the input.
Example : On input "1,4,1,4,3,1" the program should have for each $i\neq 4$ an accepting computation with output i (i.e., $i=0,1,2,3,5,6,...$). Moreover, each accepting computation of the program should provide one of these outputs.

d. The program outputs an input value v that appears as the vth value in the input.
Example : On input "3,2,1,2,5,3" the program should have an accepting computation with output "2", and an accepting computation with output "5". Moreover, each accepting computation of the program should provide either of these outputs.

e. The program outputs an input value v that appears exactly v times in the input.
Example : On input "3,2,1,2,5,3" the program should have an accepting computation with output "1", and an accepting computation with output "2". Moreover, each accepting computation of the program should provide either of these outputs.

f. The program accepts exactly those inputs whose values cannot be sorted into a sequence of consecutive numbers.
Example: The program should accept the input "1,2,1", and the input "1,4,2". On the other hand, the program should reject the input "1,3,2".

1.3.4 For each of the following cases write a program that computes the given relation.

a. $\{(x,y)\mid$ there is a value in the domain of the variables that does not appear in x and does not appear in $y\}$.
Example: With the domain of variables $\{1,2,3,4,5,6\}$ on input "1,2,4,5,6" the program can have any output that does not contain the value 3.

b. $\{(x,y)\mid x$ is not empty, and the first value in x is equal to the first value in $y\}$.

1.3.5 Let P_1 and P_2 be any two given programs. Find a program P_3 that computes the union $R(P_1)\cup R(P_2)$.

1.3.6 Let P be the program in Example 1.3.13. Give the sequence of moves between the configurations of P in the computation of P on input "1,2,1" that has the minimal number of moves.

1.4.1 Consider the following problem K_1.

Domain: $\{(a,b) \mid a,b$ are natural numbers $\}$.

Question: What is the value of the natural number c that satisfies the equality $a^2+b^2=c^2$?

Find a decision problem K_2 whose solutions are defined at the same instances as for K_1.

1.4.2 Let K be the following decision problem.

Domain: $\{(a,b,c) \mid a,b,c$ are natural numbers $\}$.

Question: Is there a pair of natural numbers x and y such that the equality $ax^2+by=c$ holds ?

a. Write a program that decides K.

b. Write a program that partially decides K, but does not decide K.

1.4.3 Show that the following problems are partially decidable for Type 0 grammars.

a. Membership problem

b. Nonemptiness problem

1.4.4 Show that the membership problem is decidable for Type 1 grammars.

1.4.5 Show that the inequivalence problem is partially decidable for Type 1 grammars.

1.4.6 Which of the following statements is correct ?

a. If the emptiness problem is decidable for Type 3 grammars, then it is also decidable for Type 0 grammars.

b. If the emptiness problem is undecidable for Type 3 grammars, then it is also undecidable for Type 0 grammars.

c. If the emptiness problem is decidable for Type 0 grammars, then it is also decidable for Type 3 grammars.

d. If the emptiness problem is undecidable for Type 0 grammars, then it is also undecidable for Type 3 grammars.

A *polynomial expression over the natural numbers*, or simply a *polynomial expression* when the natural numbers are understood, is an expression defined recursively in the following manner.

a. Each natural number is a polynomial expression of degree 0.

b. Each variable is a polynomial expression of degree 1.

c. If E_1 and E_2 are polynomial expressions of degree d_1 and d_2, respectively, then

   a. (E_1+E_2) and (E_1-E_2) are polynomial expressions of degree $\max(d_1,d_2)$.

   b. (E_1*E_2) is a polynomial expression of degree d_1+d_2.

A polynomial is called a *Diophantine polynomial* if it can be is represented by a polynomial expression, and its variables are over the natural numbers. *Hilbert's tenth problem* is the problem of determining for any given Diophantine polynomial $Q(x_1,...,x_n)$ with variables $x_1,...,x_n$ whether or not there exist $\hat{x}_1,...,\hat{x}_n$ such that $Q(\hat{x}_1,...,\hat{x}_n)=0$.

A *LOOP program* is a program that consists only of instructions of the form $x \leftarrow 0$, $x \leftarrow y$, $x \leftarrow x+1$, and **do** x α **end.** The variables can hold only natural numbers. α can be any sequence of instructions. An execution of **do** x α **end** causes the execution of α for a number of times equal to the value of x upon encountering the **do.** Each *LOOP* program has a distinct set of variables that are initialized to hold the input values. Similarly, each *LOOP* program has a distinct set of variables, called the output variables, that upon halting hold the output values of the program. Two *LOOP* programs are said to be *equivalent* if on identical input values they produce the same output values.

1.4.7 The following problems are known to be undecidable. Can you show that they are partially decidable ?

a. Hilbert's tenth problem

b. The inequivalence problem for *LOOP* programs

1.5.1 Let Ψ be the set $\{m \mid m=2^i$ for some integer $i\}$. Show that the problem of multiplication of numbers from Ψ is reducible to the problem of addition of integer numbers.

1.5.2 Show that the nonemptiness problem for programs is reducible to the acceptance problem for programs.

1.5.3 Show that Hilbert's tenth problem is reducible to the nonemptiness problem for programs.

1.5.4 Show that the problem of determining the existence of solutions over the natural numbers for systems of Diophantine equations of the following form is reducible to Hilbert's tenth problem. Each $Q_i(x_1,...,x_n)$ is assumed to be a Diophantine polynomial.

$$Q_1(x_1,...,x_n)=0$$
$$...$$
$$Q_m(x_1,...,x_n)=0$$

1.5.5 For each of the following cases show that K_1 is reducible to K_2.

a. K_1 is the emptiness problem for Type 0 grammars, and K_2 is the equivalence problem for Type 0 grammars.

b. K_1 is the membership problem for Type 0 grammars, and K_2 is the equivalence problem for Type 0 grammars.

1.7 Bibliographic Notes

The hierarchy of grammars in Section 1.2 is due to Chomsky (1959). In the classification Chomsky (1959) used an equivalent class of grammars, called context-sensitive grammars, instead of the Type 1 grammars. Type 1 grammars are due to Kuroda (1964). Harrison (1978) provides an extensive treatment for grammars and formal languages.

Nondeterminism was introduced in Rabin and Scott (1959) and applied to programs in Floyd (1967).

The study of undecidability originated in Turing (1936) and Church (1936). Hilbert's tenth problem is due to Hilbert (1901), and its undecidability to Matijasevic (1970). *LOOP* programs and the undecidability of the equivalence problem for them are due to Ritchie (1963) and Meyer and Ritchie (1967).

Chapter 2

FINITE-MEMORY PROGRAMS

Finite-memory programs are probably one of the simplest classes of programs for which our study would be meaningful. The first section of this chapter motivates the investigation of this class. The second section introduces the mathematical systems of finite-state transducers, and shows that they model the computations of finite-memory programs. The third section provides grammatical characterizations for the languages that finite-memory programs accept, and the fourth section considers the limitations of those programs. The fifth section discusses the importance of closure properties in the design of programs, and their applicability for finite-memory programs. And the last section considers properties that are decidable for finite-memory programs.

2.1 Motivation

It is often useful when developing knowledge in a new field to start by considering restricted cases and then gradually expand to the general case. Such an approach allows a gradual increase in the complexity of the argumentation. In particular, it is a quite common strategy in the investigation of infinite systems to start by considering finite subsystems. We take a similar approach here by using programs with finite domains of variables, called *finite-memory programs* or *finite-domain programs*, first.

However, it should be mentioned that finite-memory programs are also important on their own merit. They are applicable in the design and analysis of some common types of computer programs.

For instance, in compilers (i.e., in programs that translate programs written in high-level languages to equivalent programs written in machine languages) the lexical analyzers are basically designed as finite-memory programs. The main task of a lexical analyzer is to scan the given inputs and locate the symbols that belong to each of the tokens.

Example 2.1.1 Let LEXANL be the finite-memory program in Figure 2.1.1(a).

(a)

```
char := " "
do
    /* Find the first character of the next token. */
    if char = " " then
        do
            if eof then accept
            read char
        until char ≠ " "
    /* Determine the class of the token. */
    charClass := class(char)
    write className(charClass)
    /* Determine the remaining characters of the token. */
    do
        write char
        if eof then accept
        oldCharClass := charClass
        read char
        charClass := M(charClass,char)
    until charClass ≠ oldCharClass
until false
```

(b)

class

" "	0
"A"	1
⋮	⋮
"Z"	1
"0"	2
⋮	⋮
"9"	2

M

	" "	"A"	...	"Z"	"0"	...	"9"
1	0	1	⋯	1	1	⋯	1
2	0	0	⋯	0	2	⋯	2

className

1	"identifier"
2	"natural number"

Fig. 2.1.1 (a) A lexical analyzer. (b) Tables for the lexical analyzer

The domain of the variables is assumed to equal

$$\{\text{" "},\text{"}A\text{"},...,\text{"}Z\text{"},\text{"0"},...,\text{"9"},0,1,2\},$$

with " " as initial value. The functions *class*, *className*, and M are defined by the tables of Figure 2.1.1(b).

LEXANL is a lexical analyzer that determines the tokens in the given inputs, and classifies them into identifiers and natural numbers. Each identifier is represented by a letter followed by an arbitrary number of letters and digits. Each natural number is represented by one or more digits. Each pair of consecutive

tokens, except for a natural number followed by an identifier, must be separated by one or more blanks.

LEXANL can be easily modified to recognize a different class of tokens, by just redefining *class*, *className*, and M. □

Protocols for communicating processes are also examples of systems that are frequently designed as finite-memory programs. In such systems, each process is represented by a finite-memory program. Each channel from one process to another is abstracted by an implicit queue, that is, by a first-in-first-out memory. At each instance the queue holds those messages that have been sent through the channel but not received yet. Each sending of a message is represented by the writing of the message to the appropriate channel. Each receiving of a message is represented by the reading of the message from the appropriate channel.

In Section 2.6 it is shown that finite-memory programs have some interesting decidable properties. Such decidable properties make the finite-memory programs also attractive as tools for showing the complexity of some seemingly unrelated problems.

Example 2.1.2 Consider the problem K of deciding the existence of solutions over the natural numbers for systems of linear Diophantine equations, that is, for systems of equations of the following form. The a'_{ij}s and b'_is are assumed to be integers.

$$\begin{aligned} a_{11}x_1+...+a_{1n}x_n &= b_1 \\ &\vdots \\ a_{m1}x_1+...+a_{mn}x_n &= b_m \end{aligned}$$

No straightforward algorithm seems to be evident for deciding the problem, though one can easily partially decide the problem by exhaustively searching for an assignment to the variables that satisfies the given system.

For each instance I of K, a finite-memory program P_I can be constructed to accept some input if and only if I has a solution over the natural numbers. Consequently, the problem K is reducible to the emptiness problem for finite-memory programs. The decidability of K is then implied by the decidability of the emptiness problem for finite-memory programs (Theorem 2.6.1).

In fact, the proof of Theorem 2.6.1 implies that a system I has a solution over the natural numbers if and only if the system has a solution in which the values of $x_1,...,x_n$ are no greater than some bound that depends only on the a'_{ij}s, b'_is, m and n. □

Computer programs that use no auxiliary memory, except for holding the input and output values, are by definition examples of finite-memory programs.

However, such programs can deal with domains of high cardinality (i.e., 2^k for computers with k bits per word), and as a result their designs are generally not affected by the finiteness of their domains. Consequently, such programs should not generally be considered as "natural" finite-memory programs.

2.2 Finite-State Transducers

Central to the investigation of finite-memory programs is the observation that the set of all the states reachable in the computations of each such program is finite. As a result, the computations of each finite-memory program can be characterized by a finite set of states and a finite set of rules for transitions between those states.

Abstracted Finite-Memory Programs

Specifically, let P be a finite-memory program with m variables $x_1,...,x_m$, and k instruction segments $I_1,...,I_k$. Denote the initial value of the variables of P with $\odot$.

Each *state* of P is an $(m+1)$-tuple $[i,v_1,...,v_m]$, where i is an integer between 1 and k, and $v_1,...,v_m$ are values from the domain of the variables. Intuitively, a state $[i,v_1,...,v_m]$ indicates that the program reached instruction segment I_i with values $v_1,...,v_m$ in the variables $x_1,...,x_m$, respectively.

Example 2.2.1 Let P be the program in Figure 2.2.1. The domain of the variables is assumed to equal $\{0,1\}$, and the initial value is assumed to be 0. Let $[i,x,y]$ denote the state of P that corresponds to the ith instruction segment I_i, the value x in x, and the value y in y.

```
x:=?                        /* I1  */
write x                     /* I2  */
do                          /* I3  */
    do                      /* I4  */
        read y              /* I5  */
    until x=y               /* I6  */
    if eof then accept      /* I7  */
    do                      /* I8  */
        x:=x-1              /* I9  */
    or
        y:=y+1              /* I10 */
    until x≠y               /* I11 */
until false                 /* I12 */
```

Figure 2.2.1
A finite-memory program with $\{0,1\}$ as the domain of the variables

The state [1,0,0] indicates that the program reached the first instruction segment with the value 0 in x and y. The state [5,1,0] indicates that the program reached the fifth instruction segment with the value 1 in x and the value 0 in y.

From state [5,1,0] the program can reach either state [6,1,0] or state [6,1,1]. In the transition from state [5,1,0] to state [6,1,0] the program reads the value 0 and writes nothing. In the transition from state [5,1,0] to state [6,1,1] the program reads the value 1 and writes nothing. □

The computational behavior of P can be abstracted by a formal system

$$\langle Q,\Sigma,\Delta,\delta,q_0,F\rangle,$$

which is defined through the algorithm below. In the formal system

Q represents the set of states that P can reach.

δ represents the set of transitions that P can take between its states.

Σ represents the set of input values that P can read.

Δ represents the set of output values that P can write.

q_0 represents the initial state of P.

F represents the set of accepting states that P can reach.

The algorithm determines the sets Q, Σ, Δ, δ, and F by conducting a search for the elements of the sets.

Step 1 Initiate Q to the set containing just $q_0=[1,\odot,...,\odot]$, and δ to be an empty set. q_0 is called the *initial state* of P, and δ is called the *transition table* of P.

Step 2 Add the state $p=[j,u_1,...,u_m]$ of P to Q, if for some state $q=[i,v_1,...,v_m]$ in Q the following condition holds: P can, by executing I_i with values $v_1,...,v_m$ in its variables, reach I_j with $u_1,...,u_m$ in its variables, respectively.

Step 3 Add $(q,\alpha,(p,\rho))$ to δ, if P, by executing a single instruction segment, can go from state q in Q to state p in Q while reading α and writing ρ. For notational convenience, in what follows $(q,\alpha,(p,\rho))$ will be written as (q,α,p,ρ). Each tuple in δ is called a *transition rule* of P.

Step 4 Repeat Steps 2 and 3 as long as more states can be added to Q or more transition rules can be added to δ.

Step 5 Initialize Σ,Δ, and F to be empty sets.

Step 6 If (q,α,p,ρ) is a transition rule in δ and $\alpha\neq\varepsilon$ then add α to Σ. Similarly,

if (q,α,p,ρ) is a transition rule in δ and $\rho \neq \varepsilon$, then add ρ to Δ. Each α in Σ is called an *input symbol* of P, and Σ is called the *input alphabet* of P. Similarly, each ρ in Δ is called an *output symbol* of P, and Δ is called the *output alphabet* of P.

Step 7 Insert to F each state $[i,v_1,...,v_m]$ in Q for which I_i is a conditional accept instruction. The states in F are called the *accepting*, or the *final*, states of P.

By definition δ is a relation from $Q \times (\Sigma \cup \{\varepsilon\})$ to $Q \times (\Delta \cup \{\varepsilon\})$. Moreover, the sets Q,Σ,Δ,δ, and F are all finite because the number of instruction segments in P, the number of variables in P, and the domain of the variables of P are all finite.

Example 2.2.2 Assume the notations of Example 2.2.1. The initial state of the program P is [1,0,0]. By executing the first instruction, the program can move from state [1,0,0] and either enter the state [2,0,0] or the state [2,1,0]. In both cases, no input symbol is read and no output symbol is written during the transition between the states. Hence, the transition table δ for P contains the transition rules $([1,0,0],\varepsilon,[2,0,0],\varepsilon)$ and $([1,0,0],\varepsilon,[2,1,0],\varepsilon)$.

Similarly, by executing its second instruction, the program P must move from state [2,1,0] and enter state [3,1,0] while reading nothing and writing 1. Hence, δ contains also the transition rule $([2,1,0],\varepsilon,[3,1,0],1)$.

The number of states in Q is no greater than $12 \times 2 \times 2$. {0,1} is the input and the output alphabet for the program P. {[7,0,0],[7,1,1]} is the set of accepting states for P. □

Finite-State Transducers

In general, a formal system M consisting of a six-tuple $\langle Q,\Sigma,\Delta,\delta,q_0,F\rangle$ is called a *finite-state transducer* if it satisfies the following conditions.

Q is a finite set, whose members are called the *states* of M.

Σ is an alphabet, called the *input alphabet* of M. Each symbol in Σ is called an *input symbol* of M.

Δ is an alphabet, called the *output alphabet* of M. Each symbol in Δ is called an *output symbol* of M.

δ is a relation from $Q \times (\Sigma \cup \{\varepsilon\})$ to $Q \times (\Delta \cup \{\varepsilon\})$, called the *transition table* of M. Each tuple $(q,\alpha,(p,\rho))$, or simply (q,α,p,ρ), in δ is called a *transition rule* of M.

q_0 is a state in Q, called the *initial state* of M.

F is a subset of Q, whose states are called the *accepting*, or the *final*, states of M.

Example 2.2.3 The tuple

$$M = \langle \{q_0,q_1\}, \{a,b\}, \{1\}, \{(q_0,a,q_1,1),(q_0,b,q_1,\varepsilon),(q_1,b,q_1,1),(q_1,a,q_0,\varepsilon)\}, q_0, \{q_0\}\rangle$$

is a finite-state transducer. The finite-state transducer has the states q_0 and q_1. The input alphabet of M consists of two symbols a and b. The output alphabet of M consists of a single symbol 1. The finite-state transducer M has four transition rules. q_0 is the initial state of M, and the only accepting state of M.

The transition rule $(q_0,a,q_1,1)$ of M uses the input symbol a and the output symbol 1. The transition rule (q_1,a,q_0,ε) of M uses the input symbol a and no output symbol. □

Each finite-state transducer $\langle Q,\Sigma,\Delta,\delta,q_0,F\rangle$ can be graphically represented by a *transition diagram* of the following form. For each state in Q the transition diagram has a corresponding node, which is shown by a circle. The initial state is identified by an arrow from nowhere that points to the corresponding node. Each accepting state is identified by a double circle. Each transition rule (q,α,p,ρ) in δ is represented by an edge labeled with α/ρ, from the node labeled by state q to the node labeled by state p. For notational convenience edges that agree in their origin and destination are merged, and their labels are separated by commas.

Example 2.2.4 The transition diagram in Figure 2.2.2

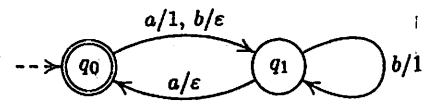


Fig. 2.2.2 Trasition diagram of a finite-state transducer

represents the finite-state transducer M of Example 2.2.3. The label $a/1$ on the edge from state q_0 to state q_1 in the transition diagram corresponds to the transition rule $(q_0,a,q_1,1)$ of M. The label b/ε on the edge from state q_0 to state q_1 corresponds to the transition rule (q_0,b,q_1,ε). The label $b/1$ on the edge from state q_1 to itself corresponds to the transition rule $(q_1,b,q_1,1)$. □

Example 2.2.5 The transition diagram in Figure 2.2.3

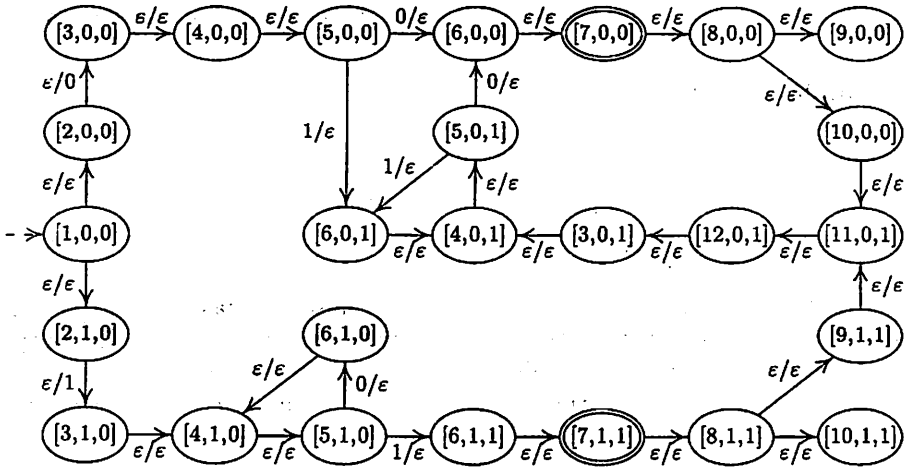


Fig. 2.2.3 Transition diagram for the program of Figure 2.2.1

represents the finite-state transducer that characterizes the program of Example 2.2.1. □

Configurations and Moves of Finite-State Transducers

Intuitively, a finite-state transducer $M=\langle Q,\Sigma,\Delta,\delta,q_0,F\rangle$ can be viewed as an abstract computing machine. The computing machine consists of a *finite-state control*, an *input tape*, a read-only *input head*, an *output tape*, and a write-only *output head* (see Figure 2.2.4).

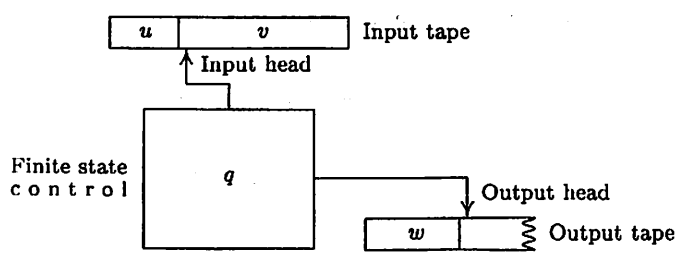


Fig. 2.2.4 A view of a finite-state transducer as an abstract computing machine

Each tape is divided into cells, which can each hold exactly one symbol.

The input tape is used for holding the input uv of M. The input head is used for accessing the input tape. The output tape is used for holding the output w of M, and the output head is used for accessing the output tape. The finite-state control is used for recording the state of M.

On each input $a_1...a_n$ from Σ^*, the computing machine M has some set of possible configurations. Each *configuration*, or *instantaneous description*, of M

is a pair (uqv,w), where q is a state in Q, $uv=a_1...a_n$, and w is a string in Δ^*. Intuitively, a configuration (uqv,w) says that M on input uv reached state q after reading u and writing w. With no loss of generality it is assumed that Q and Σ are mutually disjoint.

Example 2.2.6 Let M be the finite-state transducer of Example 2.2.3 (see Figure 2.2.2). The configuration $(aabq_1ba,1)$ of M says that M reached the state q_1 after reading $u=aab$ from the input tape and writing $w=1$ into the output tape. In addition, the configuration says that $v=ba$ is the remainder of the input (see Figure 2.2.5(a)).

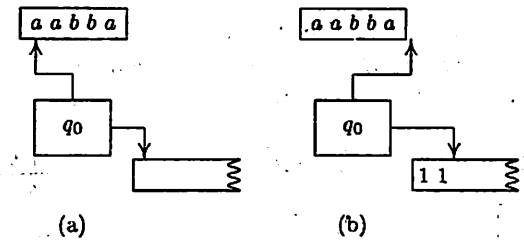


Fig. 2.2.5. Configurations of the finite-state transducer of Figure 2.2.2

The configuration (q_0aabba,ε) of M says that M reached the state q_0 after reading nothing (i.e., $u=\varepsilon$) from the input tape and writing nothing (i.e., $w=\varepsilon$) into the output tape. In addition, the configuration says that $v=aabba$ is the input to be consumed (see Figure 2.2.5(b)).

The configuration $(aabbaq_0,1)$ of M says that M reached state q_0 after reading all the input (i.e., $v=\varepsilon$) and writing $w=11$. In addition, the configuration says that the input that has been read is $u=aabba$. □

A configuration (uqv,w) of M is said to be an *initial configuration* if $q=q_0$ and $u=w=\varepsilon$. An initial configuration says that the input head is placed at the start (leftmost position) of the input, the output tape is empty, and the finite-state control is set to the initial state.

A configuration (uqv,w) of M is said to be an *accepting configuration* if $v=\varepsilon$ and q is an accepting state in F. An accepting configuration says that M reached an accepting state after reading all the input.

Example 2.2.7 The finite-state transducer M of Example 2.2.3 (see Figure 2.2.2) has the initial configuration (q_0aabba,ε), and the accepting configuration $(aabbaq_0,11)$ on input $aabba$ (see Figure 2.2.5(a) and Figure 2.2.5(b), respectively).

$(aabbaq_0,\varepsilon)$ and $(aabbaq_0,111)$ are also accepting configurations of M on input $aabba$. On the other hand, (q_0aabba,ε) is the only initial configuration of M on input $aabba$. □

The transition rules of M are used for defining the possible moves of M. Each move uses some transition rule. A *move* on transition rule (q,α,p,ρ) consists of changing the state of the finite-state control from q to p, of reading α from the input tape, of writing ρ to the output tape, and of moving the input and the output heads, $|\alpha|$ and $|\rho|$ positions to the right, respectively.

A move of M from configuration C_1 to configuration C_2 is denoted $C_1 \vdash_M C_2$, or simply $C_1 \vdash C_2$ if M is understood. A sequence of zero or more moves of M from configuration C_1 to configuration C_2 is denoted $C_1 \vdash_M^* C_2$, or simply $C_1 \vdash^* C_2$, if M is understood.

Example 2.2.8 Let M be the finite-state transducer of Example 2.2.3 (see Figure 2.2.2). On input *aabba*, M can have the following sequence $(q_0aabba,\varepsilon) \vdash^* (aabbaq_0,11)$ of moves between configurations (see Figure 2.2.6):

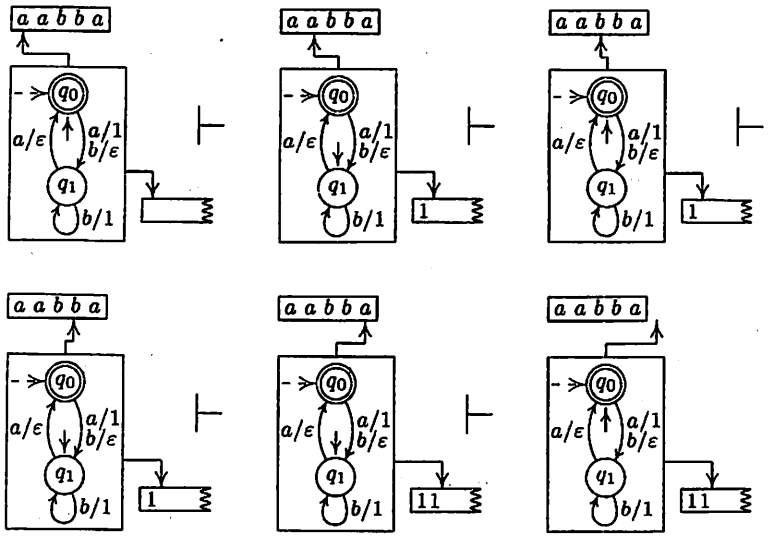


Fig. 2.2.6. Sequence of moves between configurations of a finite-state transducer

$$(q_0aabba,\varepsilon) \vdash (aq_1abba,1) \vdash (aaq_0bba,1) \vdash$$
$$(aabq_1ba,1) \vdash (aabbq_1a,11) \vdash (aabbaq_0,11).$$

The sequence consists of five moves. It starts with a move $(q_0aabba,\varepsilon) \vdash (aq_1abba,1)$ on the first transition rule $(q_0,a,q_1,1)$ of M. During the move, M makes a transition from state q_0 to state q_1 while reading a and writing 1.

The second move $(aq_1abba,1) \vdash (aaq_0bba,1)$ is on the fourth transition rule (q_1,a,q_0,ε) of M. During the move, M makes a transition from state q_1 to state q_0 while reading a and writing nothing.

The sequence continues by a move on the second transition rule (q_0,b,q_1,ε), followed by a move on the third transition rule $(q_1,b,q_1,1)$, and it terminates after an additional move on the fourth transition rule (q_1,a,q_0,ε).

The sequence of moves is the only one that can start at the initial configuration and end at an accepting configuration for the input *aabba*. □

By definition, $|\alpha|=0$ or $|\alpha|=1$ in each transition rule (q,α,p,ρ). $|\alpha|=0$ if no input symbol is read during the moves that use the transition rule (i.e., $\alpha=\varepsilon$), and $|\alpha|=1$ if exactly one input symbol is read during the moves. Similarly, $|\rho|=0$ or $|\rho|=1$, depending on whether nothing is written during the moves or exactly one symbol is written, respectively.

Determinism and Nondeterminism in Finite-State Transducers

A finite-state transducer $M=\langle Q,\Sigma,\Delta,\delta,q_0,F\rangle$ is said to be *deterministic* if, for each state q in Q and each input symbol a in Σ, the union $\delta(q,a) \cup \delta(q,\varepsilon)$ is a multiset that contains at most one element.

Intuitively, M is deterministic if each state of M fully determines whether an input symbol is to be read on a move from the state, and the state together with the input to be consumed in the move fully determine the transition rule to be used.

A finite-state transducer is said to be *nondeterministic* if the previous conditions do not hold.

Example 2.2.9 The finite-state transducer M_1, whose transition diagram is given in Figure 2.2.2, is deterministic. In each of its moves M_1 reads an input symbol. The transition rule to be used in each move is uniquely determined by the state and the input symbol being read.

If M_1 reads the input symbol a in the move from state q_0, then M_1 must use the transition rule $(q_0,a,q_1,1)$ in the move. If M_1 reads the input symbol b in the move from state q_0 then M_1 must use the transition rule (q_0,b,q_1,ε) in the move.

On the other hand, consider the finite-state transducer M_2, which satisfies

$$M_2=\langle Q,\Sigma,\Delta,\delta,q_0,F\rangle$$
for
$$Q=\{q_0,q_1,q_2,q_3\},$$
$$\Sigma=\{a,b\},$$
$$\Delta=\{a,b\},$$
$$\delta=\{(q_0,a,q_1,a),(q_1,\varepsilon,q_2,a),(q_2,\varepsilon,q_1,b),(q_2,b,q_0,\varepsilon),(q_2,b,q_3,a)\},$$
$$F=\{q_3\}.$$

The transition diagram of M_2 is given in Figure 2.2.7.

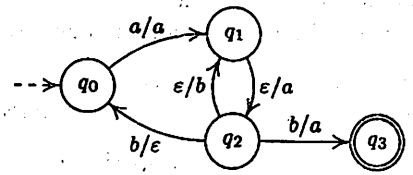


Fig. 2.2.7. A nondeterministic Finite-State transducer

M_2 is a nondeterministic finite-state transducer.

On moving from state q_0, the finite-state transducer M_2 must read an input symbol. On moving from state q_1, the finite-state transducer M_2 does not read an input symbol. The transition rules that M_2 can use on moving from states q_0 and q_1 are uniquely determined by the states, and, therefore, these states are not the source for the nondeterminism of M_2.

The source for the nondeterminism of M_2 is in the transition rules that originate at state q_2. The transition rules do not determine whether M_2 has to read a symbol in moving from state q_2, nor do they specify which of the transition rules is to be used on the moves that read the symbol b. □

Computations of Finite-State Transducers

The computations of the finite-state transducers are defined in a manner similar to that for the programs. An *accepting computation* of a finite-state transducer M is a sequence of moves of M that starts at an initial configuration and ends at an accepting configuration. A *nonaccepting*, or *rejecting*, *computation* of M is a sequence of moves on an input x for which the following conditions hold.

a. The sequence starts from the initial configuration of M on x.

b. If the sequence is finite, then it ends at a configuration from which no move is possible.

c. M has no accepting computation on x.

Each accepting computation and each nonaccepting computation of M is said to be a *computation* of M.

A computation is said to be a *halting computation* if it consists of a finite number of moves.

Example 2.2.10 Let M be the finite-state transducer of Example 2.2.3 (see Figure 2.2.2). On input *aabba* the finite-state transducer M has a computation that is given by the sequence of moves in Example 2.2.8 (see Figure 2.2.6). The computation is an accepting one.

Alternatively, on input *aab* the finite-state transducer M has the following sequence of moves:

$$(q_0aab,\varepsilon)\vdash(aq_1ab,1)\vdash(aaq_0b,1)\vdash(aabq_1,1).$$

This sequence is the only one possible from the initial configuration of M on input *abb*; it is a nonaccepting computation of M.

The two computations in the example are halting computations of M. □

By definition, on inputs that are accepted by a finite-state transducer the finite-state transducer may have also executable sequences of transition rules which are not considered to be computations.

Example 2.2.11 Consider the finite-state transducer M whose transition diagram is given in Figure 2.2.7. On input *ab*, M has the accepting computation that moves along the sequence of states q_0,q_1,q_2,q_3. Similarly, on input *ab*, M also has an accepting computation that moves along the sequence of states q_0,q_1,q_2,q_1,q_2,q_3. However, on input *ab* across the states q_0,q_1,q_2,q_0, M's sequence of moves is not a computation of M.

On input a the finite-state transducer has only one computation. The computation is a nonhalting computation that goes along the sequence of states $q_0,q_1,q_2,q_1,q_2,\ldots$ On the other hand, on input *aba* the Turing transducer has infinitely many halting computations and infinitely many nonhalting computations. All the computations on input *aba* are nonaccepting computations.

The halting computations of M on input *aba* consume just the prefix *ab* of M and move through the sequences $q_0,q_1,q_2,q_1,q_2,\ldots,q_1,q_2,q_3$ of states. The nonhalting computations of M on input *aba* consume the input until its end and move through the sequences $q_0,q_1,q_2,q_1,q_2,\ldots,q_1,q_2,q_0,q_1,q_2,q_1,q_2,\ldots$ of states. □

By definition, each move in each computation must be on a transition rule that allows the computation to eventually read all the input and thereafter reach an accepting state. Whenever more than one such alternative exists in the set of feasible transition rules, any of these alternatives can be chosen. Similarly, whenever none of the feasible transition rules satisfy the conditions above, then any of these transition rules can be chosen. This fact suggests that we view the computations of the finite-state transducers as being executed by imaginary agents with magical power.

An input x is said to be *accepted*, or *recognized*, by a finite-state transducer M if M has an accepting computation on x. An accepting computation that terminates in an accepting configuration (xq_f,y) is said to have an *output* y. The output of a nonaccepting computation is assumed to be undefined.

A finite-state transducer M is said to have an *output* y on input x if it has an

accepting computation on x with output y. M is said to *halt* on x if all the computations of M on input x are halting computations.

Example 2.2.12 The finite-state transducer M whose transition diagram is given in Figure 2.2.8

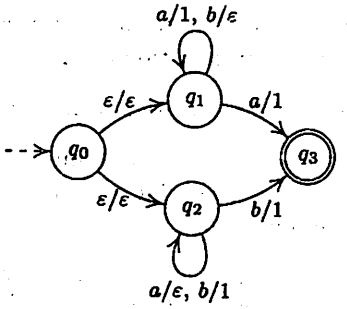


Fig. 2.2.8. A nondeterministic finite-state transducer

has, on input *baabb*, a sequence of moves that goes through the states q_0, q_1, q_1, q_1, q_1, q_1; a sequence of moves that goes through the states q_0,q_2,q_2,q_2,q_2,q_2; and a sequence of moves that goes through the states q_0,q_2,q_2,q_2,q_2,q_3. The sequence of moves that goes through the states q_0,q_2,q_2,q_2,q_2,q_3 is the only computation of M on input *baabb*. The computation is an accepting computation that provides the output 111.

M accepts all inputs. However, the finite-state transducer of Example 2.2.11 accepts exactly those inputs that have the form *ababa...bab*. □

As in the case of programs, the semantics of the finite-state transducers are characterized by their computations. Consequently, the behavior of these transducers are labeled with respect to their computations.

For instance, a finite-state transducer M is said *to move* from configuration C_1 to configuration C_2 on x if C_2 follows C_1 in the considered computation of M on x. Similarly, M is said *to read* α from its input if α is consumed from the input in the considered computation of M.

Example 2.2.13 The finite-state transducer whose transition diagram is given in Figure 2.2.8 on input *baabb* starts its computation with a move that takes M from state q_0 to state q_2. M then makes four moves, which consume *baab* and leave M in state q_2. Finally, M moves from state q_2 to state q_3 while reading b. □

Relations and Languages of Finite-State Transducers

The relation *computed* by a finite-state transducer $M=(Q,\Sigma,\Delta,\delta,q_0,F)$, denoted $R(M)$, is the set

$$\{(x,y)\,|\,(q_0x,\varepsilon)\vdash^*(xq_f,y) \text{ for some } q_f \text{ in } F\}.$$

That is, the relation computed by M is the set of all the pairs (x,y) such that M has an accepting computation on input x with output y.

The language *accepted*, or *recognized*, by M, denoted $L(M)$, is the set of all the inputs that M accepts, that is, the set

$$\{x\,|\,(x,y) \text{ is in } R(M) \text{ for some } y\}.$$

The language is said to be *decided* by M if, in addition, M halts on all inputs, that is, on all x in Σ^*.

The language *generated* by M is the set of all the outputs that M has on its inputs, that is, the set

$$\{y\,|\,(x,y) \text{ is in } R(M) \text{ for some } x\}.$$

Example 2.2.14 The nondeterministic finite-state transducer M whose transition diagram is given in Figure 2.2.8 computes the relation

$$R(M)=\{(x,1^i)\,|\,x \textit{ is in } \{a,b\}^*,\ i=\textit{number of } a\textit{'s in } x \textit{ if the last symbol in } x \textit{ is } a,\ \textit{and } i=\textit{number of } b\textit{'s in } x \textit{ if the last symbol in } x \textit{ is } b\}.$$

The finite-state automaton M accepts the language $L(M)=\{a,b\}^*$. □

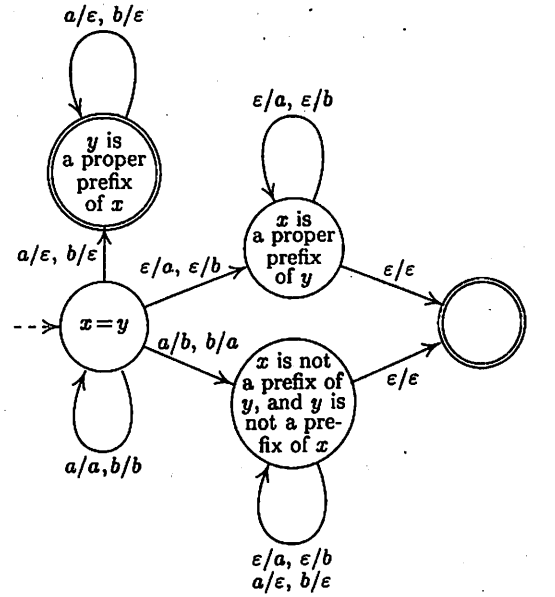


Fig. 2.2.9. A finite-state transducer that computes the relation $R(M)=\{(x,y)\,|\,x \text{ and } y \text{ are in } \{a,b\}^*, \text{ and } y\neq x\}$

Example 2.2.15 The nondeterministic finite-state transducer M whose transition diagram is given in Figure 2.2.9 computes the relation

$$R(M) = \{(x,y) \mid x \text{ and } y \text{ are in } \{a,b\}^*, \text{ and } y \neq x\}.$$

As long as M is in its initial state "$x=y$" the output of M is equal to the portion of the input consumed so far.

If M wants to provide an output that is a proper prefix of its input, then upon reaching the end of the output, M must move from the initial state to state "y is proper prefix of x."

If M wants its input to be a proper prefix of its output, then M must move to state "x is a proper prefix of y" upon reaching the end of the input.

Otherwise, at some nondeterministically chosen instance of the computation, M must move to state "x is not a prefix of y, and y is not a prefix of x," to create a discrepancy between a pair of corresponding input and output symbols. □

From Finite-State Transducers to Finite-Memory Programs

The previous discussion shows us that there is an algorithm that translates any given finite-memory program into an equivalent finite-state transducer, that is, into a finite-state transducer that computes the same relation as the program. Conversely, there is also an algorithm that derives an equivalent finite-memory program from any given finite-state transducer. The program can be a "table-driven" program that simulates a given finite-state transducer $M = \langle Q,\Sigma,\Delta,\delta,q_0,F\rangle$ in the manner described in Figure 2.2.10.

```
state := q0
do
    /* Accept if an accepting state of M is reached at the end of the input. */
    if F(state) then
        if eof then accept
    /* Nondeterministically find the entries of the transition rule
    (q,α,p,ρ) used in the next simulated move. */
    do in := e or read in until true        /* in := α */
    next_state := ?                         /* next_state := p */
    out := ?                                /* out := ρ */
    if not δ(state,in,next_state,out) then reject
    /* Simulate the move. */
    if out ≠ e then
        write out
    state := next_state
until false
```

Figure 2.2.10 A table-driven finite-memory program for simulating a finite-state transducer

The program uses a variable *state* for recording M's state in a given move, a variable *in* for recording the input M consumes in a given move, a variable *next_state* for recording the state M enters in a given move, and a variable *out* for recording the output M writes in a given move.

The program starts a simulation of M by initializing the variable *state* to the initial state q_0 of M. Then M enters an infinite loop.

The program starts each iteration of the loop by checking whether an accepting state of M has been reached at the end of the input. If such is the case, the program halts in an accepting configuration. Otherwise, the program simulates a single move of M. The predicate F is used to determine whether *state* holds an accepting state.

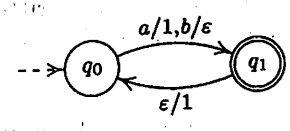
The simulation of each move of M is done in a nondeterministic manner. The program guesses the value for variable *in* that has to be read in the simulated move, the state for variable *next_state* that M enters in the simulated move, and the value for variable *out* that the program writes in the simulated move. Then the program uses the predicate δ to verify that the guessed values are appropriate and continues according to the outcome of the verification

The domain of the variables of the program is assumed to equal $Q \cup \Sigma \cup \Delta \cup \{e\}$, where e is assumed to be a new symbol not in $Q \cup \Sigma \cup \Delta$, used for denoting the empty string ε.

In the table-driven program, F is a predicate that assumes a true value when, and only when, its parameter is an accepting state. Similarly, δ is a predicate that assumes a true value when, and only when, its entries correspond to a transition rule of M.

The programs that correspond to different finite-state transducers differ in the domains of their variables and in the truth assignments for the predicates F and δ.

The algorithm can be easily modified to give a deterministic finite-memory program whenever the finite-state transducer M is deterministic.



(a)

F	
q_0	false
q_1	true

δ	q_0,e	$q_0,1$	q_1,e	$q_1,1$
q_0,e	false	false	false	false
q_0,a	false	false	false	true
q_0,b	false	false	true	false
q_1,e	false	true	false	false
q_1,a	false	false	false	false
q_1,b	false	false	false	false

(b)

δ_{in}	
q_0	true
q_1	false

δ_{out}	e	a	b
q_0		1	e
q_1	1		

δ_{state}	e	a	b
q_0		q_1	q_1
q_1	q_0		

(c)

Fig. 2.2.11. (a) A finite-state transducer M. (b) Tables for a table-driven program that simulates M. (c) Tables for a deterministic table-driven program that simulates M

Example 2.2.16 For the finite-state transducer M of Figure 2.2.11(a), the program in Figure 2.2.10 has the domain of variables $\{a,b,1,q_0,q_1,e\}$. The truth values of the predicates F and δ are defined by the corresponding tables of Figure 2.2.11(b).

The program also allows that for F and δ there are parameters that differ from those specified in the tables. On such parameters the predicates are assumed to be undefined.

The finite-state transducer can be simulated also by the deterministic table-driven program in Figure 2.2.12.

```
state := q0
do
    if F(state) then
        if eof then accept
    if not δin(state) then
        in := e
    if δin(state) then
        read in
    next_state := δstate(state,in)
    out := δout(state,in)
    if out ≠ e then
        write out
    state := next_state
until false
```

Figure 2.2.12 A table-driven, deterministic finite-memory program for simulating a deterministic finite-state transducer.

F is assumed to be a predicate as before, and δ_{in},δ_{out}, and δ_{state} are assumed to be defined by the corresponding tables in Figure 2.2.11(c).

The predicate δ_{in} determines whether an input symbol is to be read on moving from a given state. The function δ_{out} determines the output to be written in each simulated move, and δ_{state} determines the state to be reached in each simulated state.

The deterministic finite-state transducer can be simulated also by a non-table-driven finite-memory program of the form shown in Figure 2.2.13.


```
state := q0
do
    if state = q0 then
    do
        read in
        if in = a then
        do
            state := q1
            out := 1
            write out
        until true
        if in = b then
            state := q1
    until true
    if state = q1 then
    do
        if eof then accept
        state := q0
        out := 1
        write out
    until true
until false
```

Figure 2.2.13 A non-table-driven deterministic finite-memory program that simulates the deterministic finite-state transducer of Figure 2.2.11(a).

In such a case, through conditional if instructions, the program explicitly records the effect of F, δ_{in}, δ_{out}, and δ_{state}.

It follows that the finite-state transducers characterize the finite-memory programs, and so they can be used for designing and analyzing finite-memory programs. As a result, the study conducted below for finite-state transducers applies also for finite-memory programs.

Finite-state transducers offer advantages in

a. Their straightforward graphic representations, which are in many instances more "natural" than finite-memory programs.

b. Their succinctness, because finite-state transducers are abstractions that ignore those details irrelevant to the study undertaken.

c. The close dependency of the outputs on the inputs.

2.3 Finite-State Automata and Regular Languages

The computations of programs are driven by their inputs. The outputs are just the results of the computations, and they have no influence on the course that the computations take. Consequently, it seems that much can be studied about finite-state transducers, or equivalently, about finite-memory programs even when their outputs are ignored. The advantage of conducting a study of such stripped-down finite-state transducers is in the simplified argumentation that they allow.

Finite-State Automata

A finite-state transducer whose output components are ignored is called a finite-state automaton. Formally, a *finite-state automaton* M is a tuple $\langle Q,\Sigma,\delta,q_0,F\rangle$, where Q, Σ, q_0, and F are defined as for finite-state transducers, and the transition table δ is a relation from $Q\times(\Sigma\cup\{\varepsilon\})$ to Q.

Transition diagrams similar to those used for representing finite-state transducers can also be used to represent finite-state automata. The only difference is that in the case of finite-state automata, an edge that corresponds to a transition rule (p,α,p) is labeled by the string α.

Example 2.3.1 The finite-state automaton that is induced by the finite-state transducer of Figure 2.2.2 is

$$\langle Q,\Sigma,\delta,q_0,F\rangle,$$
where
$$Q=\{q_0,q_1\},$$
$$\Sigma=\{a,b\},$$
$$\delta=\{(q_0,a,q_1),(q_0,b,q_1),(q_1,b,q_1),(q_1,a,q_0)\},$$
$$F=\{q_0\}.$$

The transition diagram in Figure 2.3.1 represents the finite-state automaton. □

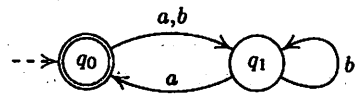


Fig. 2.3.1. A finite-state automaton that corresponds to the finite-state transducer of Figure 2.2.2

The finite-state automaton M is said to be *deterministic* if, for each state q in Q and for each input symbol a in Σ, the union $\delta(q,a)\cup\delta(q,\varepsilon)$ is a multiset

that contains at most one element. The finite-state automaton is said to be *nondeterministic* if it is not a deterministic finite-state automaton.

A transition rule (q,α,p) of the finite-state automaton is said to be an *ε transition rule* if $\alpha=\varepsilon$. A finite-state automaton with no ε transition rules is said to be an *ε-free* finite-state automaton.

Example 2.3.2 Consider the finite-state automaton

$$M_1=\langle\{q_0,\ldots,q_6\},\{0,1\},$$
$$\{(q_0,0,q_0),(q_0,\varepsilon,q_1),(q_0,\varepsilon,q_4),(q_1,0,q_2),(q_1,1,q_1),$$
$$(q_2,0,q_3),(q_2,1,q_2),(q_3,0,q_3),(q_3,1,q_1),(q_4,0,q_4),$$
$$(q_4,1,q_5),(q_5,0,q_5),(q_5,1,q_6),(q_6,1,q_6),(q_6,0,q_4)\},$$
$$q_0,\{q_0,q_3,q_6\}\rangle.$$

The transition diagram of M_1 is given in Figure 2.3.2.

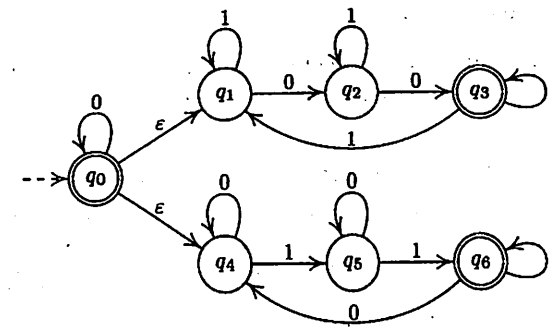


Fig. 2.3.2. A nondeterministic finite-state automaton

M_1 is nondeterministic owing to the transition rules that originate at state q_0. One of the transition rules requires that an input value be read, whereas the other two transition rules require that no input value be read. Moreover, M_1 is also nondeterministic when the transition rule $(q_0,0,q_0)$ is ignored, because M_1 cannot determine locally which of the other transition rules to follow on the moves that originate at state q_0.

The finite-state automaton M_2 in Figure 2.3.3 is a deterministic finite-state automaton.

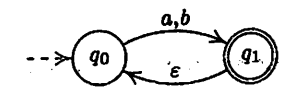


Fig. 2.3.3. A deterministic finite-state automaton

M_1 has two ε transition rules, and M_2 has one. □

A *configuration*, or an *instantaneous description*, of the finite-state automaton is a singleton uqv, where q is a state in Q, and uv is a string in Σ^*. The configuration is said to be an *initial configuration* if $u=\varepsilon$ and q is the initial state. The configuration is said to be an *accepting*, or *final*, *configuration* if $v=\varepsilon$ and q is an accepting state. With no loss of generality it is assumed that Q and Σ are mutually disjoint.

Other definitions, like those of $\vdash_M$, $\vdash$, $\vdash_M^*$, $\vdash^*$, and acceptance, recognition, and decidability of a language by a finite-state automaton, are similar to those given for finite-state transducers.

Nondeterminism versus Determinism in Finite-State Automata

By the following theorem, nondeterminism does not add to the recognition power of finite-state automata, even though it might add to their succinctness. The proof of the theorem provides an algorithm for constructing, from any given n-state finite-state automaton, an equivalent deterministic finite-state automaton of at most 2^n states.

Theorem 2.3.1 If a language is accepted by a finite-state automaton, then it is also decided by a deterministic finite-state automaton that has no ε transition rules.

Proof Consider any finite-state automaton $M=\langle Q,\Sigma,\delta,q_0,F\rangle$. Let A_x denote the set of all the states that M can reach from its initial state q_0, by the sequences of moves that consume the string x, that is, the set $\{q \mid q_0 x \vdash^* xq\}$. Then an input w is accepted by M if and only if A_w contains an accepting state.

The proof relies on the observation that A_{xa} contains exactly those states that can be reached from the states in A_x, by the sequences of transition rules that consume a, that is, $A_{xa}=\{p \mid q \text{ is in } A_x, \text{ and } qa \vdash^* ap\}$.

Specifically, if p is a state in A_{xa}, then by definition there is a sequence of transition rules $\tau_1,...,\tau_t$ that takes M from the initial state q_0 to state p while consuming xa. This sequence must have a prefix $\tau_1,...,\tau_i$ that takes M from q_0 to some state q while consuming x (see Figure 2.3.4(a)).

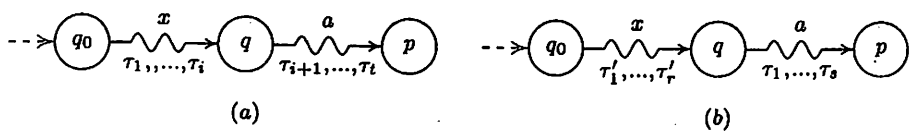


Fig. 2.3.4. Sequences of transition rules that consume xa

Consequently, q is in A_x and the subsequence $\tau_{i+1},...,\tau_t$ of transition rules takes

M from state q to state p while consuming a.

On the other hand, if q is in A_x and if p is a state that is reachable from state q by a sequence $\tau_1,...,\tau_s$ of transition rules that consumes a, then the state p is in A_{xa}. In such a case, if $\tau'_1,...,\tau'_r$ is a sequence of transition rules that takes M from the initial state q_0 to state q while consuming x, then M can reach the state p from state q_0 by the sequence $\tau'_1,...,\tau'_r,\tau_1,...,\tau_s$ of transition rules that consumes xa (see Figure 2.3.4(b)).

As a result, to determine if $a_1...a_n$ is accepted by M, one needs only to follow the sequence $A_\varepsilon, A_{a_1}, A_{a_1a_2},..., A_{a_1...a_n}$ of sets of states, where each $A_{a_1...a_{i+1}}$ is uniquely determined from $A_{a_1...a_i}$ and a_{i+1}. Therefore, a deterministic finite-state automaton M' of the following form decides the language that is accepted by M.

The set of states of M' is equal to

$$\{A \mid A \text{ is a subset of } Q, \text{ and } A=A_x \text{ for some } x \text{ in } \Sigma^*\}.$$

Since Q is finite, it follows that Q has only a finite number of subsets A, and consequently M' has also only a finite number of states. The initial state of M' is the subset of Q that is equal to A_ε. The accepting states of M' are those states of M' that contain at least one accepting state of M. The transition table of M' is the set

$$\{(A,a,A') \mid A \text{ and } A' \text{ are states of } M', a \text{ is in } \Sigma, \text{ and } A' \text{ is the set of states that the finite-state automaton } M \text{ can reach by consuming } a \text{ from those states that are in } A\}.$$

By definition, M' has no ε transition rules. Moreover, M' is deterministic because, for each x in Σ^* and each a in Σ, the set A_{xa} is uniquely defined from the set A_x and the symbol a. □

Example 2.3.3 Let M be the finite-state automaton whose transition diagram is given in Figure 2.3.2. The transition diagram in Figure 2.3.5

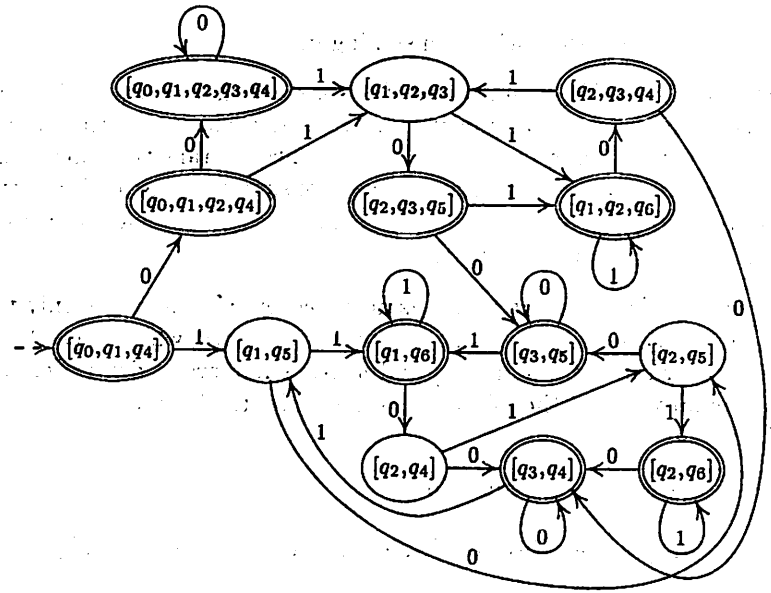


Fig. 2.3.5. A transition diagram of an ε-free, deterministic finite-state automaton that is equivalent to the finite-state automaton whose transition diagram is given in Figure 2.3.2

represents an ε-free, deterministic finite-state automaton that is equivalent to M. Using the terminology of the proof of Theorem 2.3.1

$$A_\varepsilon = \{q_0,q_1,q_4\},$$
$$A_0 = \{q_0,q_1,q_2,q_4\}, \text{ and}$$
$$A_{00} = A_{000} = \ldots = A_{0\ldots0} = \{q_0,q_1,q_2,q_3,q_4\}.$$

A_ε is the set of all the states that M can reach without reading any input. q_0 is in A_ε because it is the initial state of M. q_1 and q_2 are in A_ε because M has ε transition rules that leave the initial state q_0 and enter states q_1 and q_2, respectively.

A_0 is the set of all the states that M can reach just by reading 0 from those states that are in A_ε. q_0 is in A_0 because q_0 is in A_ε and M has the transition rule $(q_0,0,q_0)$. q_1 is in A_0 because q_0 is in A_ε and M can use the pair $(q_0,0,q_0)$ and (q_0,ε,q_1) of transition rules to reach q_1 from q_0 just by reading 0. q_2 is in A_0 because q_0 is in A_ε and M can use the pair (q_0,ε,q_1) and $(q_1,0,q_2)$ of transition rules to reach q_2 from q_0 just by reading 0. □

The result of the last theorem cannot be generalized to finite-state transducers, because deterministic finite-state transducers can only compute functions, whereas nondeterministic finite-state transducers can also compute relations which are not functions, for example, the relation $\{(a,b),(a,c)\}$. In fact, there

are also functions that can be computed by nondeterministic finite-state transducers but that cannot be computed by deterministic finite-state transducers. The following relation

$$R = \{(x0,0^{|x|}) \mid x \text{ is a string in } \{0,1\}^*\} \cup \{(x1,1^{|x|}) \mid x \text{ is a string in } \{0,1\}^*\}$$

is an example of such a function. The function cannot be computed by a deterministic finite-state transducer because each deterministic finite-state transducer M satisfies the following condition, which is not shared by the function: if x_1 is a prefix of x_2 and M accepts x_1 and x_2, then the output of M on input x_1 is a prefix of the output of M on input x_2 (Exercise 2.2.5).

Finite-State Automata and Type 3 Grammars

The following two results imply that a language is accepted by a finite-state automaton if and only if it is a Type 3 language. The proof of the first result shows how Type 3 grammars can simulate the computations of finite-state automata.

Theorem 2.3.2 Finite-state automata accept only Type 3 languages.

Proof Consider any finite-state automaton $M = \langle Q,\Sigma,\delta,q_0,F\rangle$. By Theorem 2.3.1 it can be assumed that M is an ε-free, finite-state automaton. With no loss of generality, it can also be assumed that no transition rule takes M to its initial state when that state is an accepting one. (If such is not the case, then one can add a new state q_0' to Q, make the new state q_0' both an initial and an accepting state, and add a new transition rule (q_0',α,q) to δ for each transition rule of the form (q_0,α,q) that is in δ.)

Let $G = \langle N,\Sigma,P,\ [q_0]\rangle$ be a Type 3 grammar, where N has a nonterminal symbol $[q]$ for each state q in Q and P has the following production rules.

a. A production rule of the form $[q] \to a[p]$ for each transition rule (q,a,p) in the transition table δ.

b. A production rule of the form $[q] \to a$ for each transition rule (q,a,p) in δ such that p is an accepting state in F.

c. A production rule of the form $[q_0] \to \varepsilon$ if the initial state q_0 is an accepting state in F.

The grammar G is constructed to simulate the computations of the finite-state automaton M. G records the states of M through the nonterminal symbols. In particular, G uses its start symbol $[q_0]$ to initiate a simulation of M at state q_0. G uses a production rule of the form $[q] \to a[p]$ to simulate a move of M from state q to state p. In using such a production rule, G generates the symbol a that M reads in the corresponding move. G uses a production rule of the form

$[q] \rightarrow$ a instead of the production rule of the form $[q] \rightarrow a[p]$, when it wants to terminate a simulation at an accepting state p.

By induction on n it follows that a string $a_1 a_2 ... a_n$ has a derivation in G of the form

$$[q] \Rightarrow a_1[q_1] \Rightarrow a_1 a_2[q_2] \Rightarrow ... \Rightarrow a_1 a_2 ... a_{n-1}[q_{n-1}] \Rightarrow a_1 a_2 ... a_n$$

if and only if M has a sequence of moves of the form

$$qa_1 a_2 ... a_n \vdash a_1 q_1 a_2 ... a_n \vdash a_1 a_2 q_2 a_3 ... a_n \vdash ... \vdash a_1 ... a_{n-1} q_{n-1} a_n \vdash a_1 a_2 ... a_n q_n$$

for some accepting state q_n. In particular the correspondence above holds for $q = q_0$. Therefore $L(G) = L(M)$. □

Example 2.3.4 The finite-state automaton M_1, whose transition diagram is given in Figure 2.3.6(b),

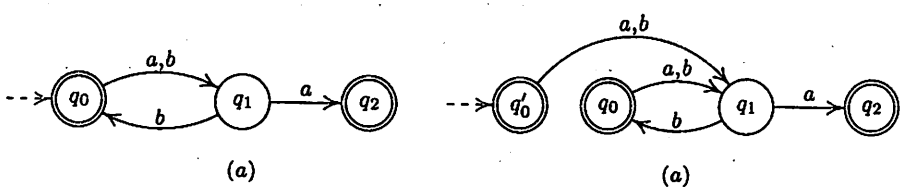


Fig. 2.3.6. Two equivalent finite-state automata

is an ε-free, deterministic finite-state automaton. M_1 is not suitable for a direct simulation by a Type 3 grammar because its initial state q_0 is both an accepting state and a destination of a transition rule. Without modifications to M_1 the algorithm that constructs the grammar G will produce the production rule $[q_0] \rightarrow \varepsilon$ because q_0 is an accepting state, and the production rule $[q_1] \rightarrow b[q_0]$ because of the transition rule (q_1, b, q_0). Such a pair of production rules cannot coexist in a Type 3 grammar.

M_1 is equivalent to the finite-state automaton M_2, whose transition diagram is given in Figure 2.3.6(a). The Type 3 grammar $G = \langle N, \Sigma, P, [q_0'] \rangle$ generates the language $L(M_2)$, if $N = \{[q_0'], [q_0], [q_1], [q_2]\}$, $\Sigma = \{a, b\}$, and P consists of the following production rules:

$$\begin{aligned}
[q_0'] &\rightarrow \varepsilon \\
&\rightarrow a[q_1] \\
&\rightarrow b[q_1] \\
[q_0] &\rightarrow a[q_1] \\
&\rightarrow b[q_1] \\
[q_1] &\rightarrow b[q_0] \\
&\rightarrow b \\
&\rightarrow a[q_2] \\
&\rightarrow a
\end{aligned}$$

The accepting computation

$$q_0' abaa \vdash aq_1 baa \vdash abq_0 aa \vdash abaq_1 a \vdash abaaq_2$$

of M_2 on input $abaa$ is simulated by the derivation

$$[q_0'] \Rightarrow a[q_1] \Rightarrow ab[q_0] \Rightarrow aba[q_1] \Rightarrow abaa$$

of the grammar.

The production rule $[q_1] \rightarrow a[q_2]$ can be eliminated from the grammar without affecting the generated language. □

The next theorem shows that the converse of Theorem 2.3.2 also holds. The proof shows how finite-state automata can trace the derivations of Type 3 grammars.

Theorem 2.3.3 Each Type 3 language is accepted by a finite-state automaton.

Proof Consider any Type 3 grammar $G = \langle N, \Sigma, P, S \rangle$. The finite-state automaton $M = \langle Q, \Sigma, \delta, q_S, F \rangle$ accepts the language that G generates if Q, δ, q_S, and F are as defined below.

M has a state q_A in Q for each nonterminal symbol A in N. In addition, Q also has a distinguished state named q_f. The state q_S of M, which corresponds to the start symbol S, is designated as the initial state of M. The state q_f of M is designated to be the only accepting state of M, that is, $F = \{q_f\}$.

M has a transition rule in δ if and only if the transition rule corresponds to a production rule of G. Each transition rule of the form (q_A, a, q_B) in δ corresponds to a production rule of the form $A \rightarrow aB$ in G. Each transition rule of the form (q_A, a, q_f) in δ corresponds to a production rule of the form $A \rightarrow$ a in G. Each transition rule of the form (q_S, ε, q_f) in δ corresponds to a production rule of the form $S \rightarrow \varepsilon$ in G.

The finite-state automaton M is constructed so as to trace the derivations of the grammar G in its computations. M uses its states to keep track of the nonterminal symbols in use in the sentential forms of G. M uses its transition rules to consume the input symbols that G generates in the direct derivations that use the corresponding production rules.

By induction on n, the constructed finite-state automaton M has a sequence

$$q_{A_0} x \vdash u_1 q_{A_1} v_1 \vdash u_2 q_{A_2} v_2 \vdash ... \vdash u_{n-1} q_{A_{n-1}} v_{n-1} \vdash x q_{A_n}$$

of n moves if and only if the grammar G has a derivation of length n of the form

$$A_0 \Rightarrow u_1 A_1 \Rightarrow u_2 A_2 \Rightarrow ... \Rightarrow u_{n-1} A_{n-1} \Rightarrow x.$$

In particular, such correspondence holds for $A_0 = S$. Consequently, x is in $L(M)$ if and only if it is in $L(G)$. □

Example 2.3.5 Consider the Type 3 grammar $G = \langle \{S,A,B\}, \{a,b\}, P, S \rangle$, where P consists of the following transition rules.

$$\begin{aligned} S &\to \varepsilon \\ &\to aA \\ &\to bB \\ A &\to aA \\ &\to b \\ B &\to bB \\ &\to a \end{aligned}$$

The transition diagram in Figure 2.3.7

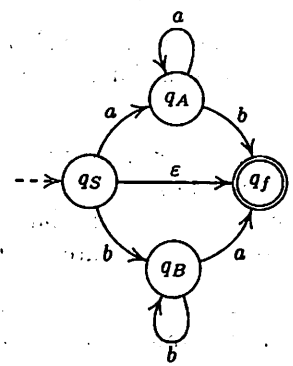


Fig. 2.3.7. A finte-state automaton that acepts $L(G)$, where G is the grammar of example 2.3.5

represents a finite-state automaton that accepts the language $L(G)$. The derivation

$$S \Rightarrow aA \Rightarrow aaA \Rightarrow aab$$

in G is traced by the computation

$$q_S aab \vdash aq_A ab \vdash aaq_A b \vdash aabq_f$$

of M. □

It turns out that finite-state automata and Type 3 grammars are quite similar mathematical systems. The states in the automata play a role similar to the nonterminal symbols in the grammars, and the transition rules in the automata play a role similar to the production rules in the grammars.

Type 3 Grammars and Regular Grammars

Type 3 grammars seem to be minimal in the sense that placing further meaningful restrictions on them results in grammars that cannot generate all the Type 3 languages. On the other hand, some of the restrictions placed on Type 3 grammars can be relaxed without increasing the class of languages that they can generate.

Specifically, a grammar $G = \langle N, \Sigma, P, S \rangle$ is said to be a *right-linear grammar* if each of its production rules is either of the form $A \to xB$ or of the form $A \to x$, where A and B are nonterminal symbols in N and x is a string of terminal symbols in Σ^*.

The grammar is said to be a *left-linear grammar* if each of its production rules is either of the form $A \to Bx$ or of the form $A \to x$, where A and B are nonterminal symbols in N and x is a string of terminal symbols in Σ^*.

The grammar is said to be a *regular grammar* if it is either a right-linear grammar or a left-linear grammar. A language is said to be a *regular language* if it is generated by a regular grammar.

By Exercise 2.3.5 a language is a Type 3 language if and only if it is regular.

Regular Languages and Regular Expressions

Regular languages can also be defined, from the empty set and from some finite number of singleton sets, by the operations of union, composition, and Kleene closure. Specifically, consider any alphabet Σ. Then a *regular set* over Σ is defined in the following way.

a. The empty set $\varnothing$, the set $\{\varepsilon\}$ containing only the empty string, and the set $\{a\}$ for each symbol a in Σ, are regular sets.

b. If L_1 and L_2 are regular sets, then so are the union $L_1 \cup L_2$, the composition $L_1 L_2$, and the Kleene closure L_1^*.

c. No other set is regular.

By Exercise 2.3.6 the following characterization holds.

Theorem 2.3.4 A set is a regular set if and only if it is accepted by a finite-state automaton.

Regular sets of the form $\varnothing$, $\{\varepsilon\}$, $\{a\}$, $L_\alpha \cup L_\beta$, $L_\alpha L_\beta$, and L_α^* are quite often denoted by the expressions $\varnothing$, ε, a, $(\alpha)+(\beta)$, $(\alpha)(\beta)$, and $(\alpha)^*$, respectively. α and β are assumed to be the expressions that denote L_α and L_β in a similar manner, respectively. a is assumed to be a symbol from the alphabet. Expressions that denote regular sets in this manner are called *regular expressions*.

Some parentheses can be omitted from regular expressions, if a precedence relation between the operations of

(1) Kleene closure,

(2) composition, *and*

(3) union

in the given order is assumed. The omission of parentheses in regular expressions is similar to that in arithmetic expressions, where closure, composition, and union in regular expressions play a role similar to exponentiation, multiplication, and addition in arithmetic expressions.

Example 2.3.6 The regular expression

$$0^*(1^*01^*00^*(11^*01^*00^*)^*+0^*10^*11^*(00^*10^*11^*)^*)$$

denotes the language that is recognized by the finite-state automaton whose transition diagram is given in Figure 2.3.2. The expression indicates that each string starts with an arbitrary number of 0's. Then the string continues with a string in $1^*01^*00^*(11^*01^*00^*)^*$ or with a string in $10^*11^*(00^*10^*11^*)^*$. In the first case, the string continues with an arbitrary number of 1's, followed by 0, followed by an arbitrary number of 1's, followed by one or more 0's, followed by an arbitrary number of strings in $11^*01^*00^*$. □

By the previous discussion, nondeterministic finite-state automata, deterministic finite-state automata, regular grammars, and regular expressions are all characterizations of the languages that finite-memory programs accept. Moreover, there are effective procedures for moving between the different characterizations. These procedures provide the foundation for many systems that produce finite-memory-based programs from characterizations of the previous nature. For instance, one of the best known systems, called LEX, gets inputs that are generalizations of regular expressions and provides outputs that are scanners. The advantage of such systems is obviously in the reduced effort they require for obtaining the desired programs.

Figure 2.3.8 illustrates the structural and functional hierarchies for some descriptive systems. The structural hierarchies are shown by the directed acyclic graphs. The functional hierarchy is shown by the Venn diagram.

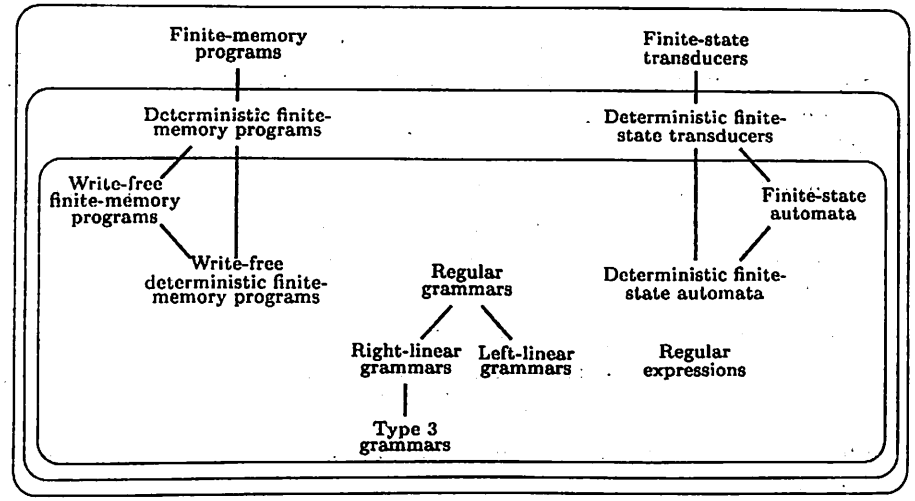


Fig. 2.3.8. The structural and functional relationship between some descriptive systems

2.4 Limitations of Finite-Memory Programs

It can be intuitively argued that there are computations that finite-memory programs cannot carry out, because of the limitations imposed on the amount of memory the programs can use. For instance, it can be argued that $\{a^nb^n \mid n \geqslant 0\}$ is not recognizable by any finite-memory program. The reasoning here is that upon reaching the first b in a given input, the program must remember how many a's it read. Moreover, the argument continues that each finite-memory program has an upper bound on the number of values that it can record, whereas no such bound exists on the number of a's that the inputs can contain. As a result, one can conclude that each finite-memory program can recognize only a finite number of strings in the set $\{a^nb^n \mid n \geqslant 0\}$.

The purposes of this section are to show that there are computations that cannot be carried out by finite-memory programs, and to provide formal tools for identifying such computations. The proofs rely on abstractions of the intuitive argument above. However, it should be mentioned that the problem of determining for any given language, whether the language is recognizable by a finite-memory program, can be shown to be undecidable (see Theorem 4.5.6). Therefore, no tool can be expected to provide an algorithm that decides the problem in its general form.

A Pumping Lemma for Regular Languages

The following theorem provides necessary conditions for a language to be de-

cidable by a finite-memory program. The proof of the theorem relies on the observations that the finite-memory programs must repeat a state on long inputs, and that the subcomputations between the repetitions of the states can be pumped.

Theorem 2.4.1 (Pumping lemma for regular languages) Every regular language L has a number m for which the following conditions hold. If w is in L and $|w| \geqslant m$, then w can be written as xyz, where xy^kz is in L for each $k \geqslant 0$. Moreover, $|xy| \leqslant m$, and $|y| > 0$.

Proof Consider any regular language L. Let M be a finite-state automaton that recognizes L. By Theorem 2.3.1 it can be assumed that M has no ε transition rules. Denote by m the number of states of M.

On input $w = a_1...a_n$ from L the finite-state automaton M has a computation of the form

$$\begin{aligned} p_0a_1...a_n &\vdash a_1p_1a_2...a_n \\ &\vdash ... \\ &\vdash a_1...a_ip_ia_{i+1}...a_n \\ &\vdash ... \\ &\vdash a_1...a_jp_ja_{j+1}...a_n \\ &\vdash ... \\ &\vdash a_1...a_np_n \end{aligned}$$

The computation goes through some sequence $p_0, p_1, ..., p_n$ of $n+1$ states, where p_0 is the initial state of M and p_n is an accepting state of M. In each move of the computation exactly one input symbol is being read.

If the length n of the input is equal at least to the number m of states of M, then the computation consists of m or more moves and some state q must be repeated within the first m moves. That is, if $n \geqslant m$ then $p_i = p_j$ for some i and j such that $0 \leqslant i < j \leqslant m$. In such a case, take $x = a_1...a_i$, $y = a_{i+1}...a_j$, and $z = a_{j+1}...a_n$.

With such a decomposition xyz of w the above computation of M takes the form

$$p_0xyz \vdash^* xqyz \vdash^* xyqz \vdash^* xyzp_n$$

During the computation the state $q = p_i = p_j$ of M is repeated. The string x is consumed before reaching the state q that is repeated. The string y is consumed between the repetition of the state q. The string z is consumed after the repetition of state q.

Consequently, M also has an accepting computation of the form

$$\begin{aligned} p_0xy^kz &\vdash^* xqy^kz \\ &\vdash^* xyqy^{k-1}z \\ &\vdash ... \\ &\vdash xy^kqz \\ &\vdash xy^kzp_n \end{aligned}$$

for each $k \geqslant 0$. That is, M has an accepting computation on xy^kz for each $k \geqslant 0$, where M starts and ends consuming each y in state q.

The substring y that is consumed between the repetition of state q is not empty, because by assumption M has no ε transition rules. □

Example 2.4.1 Let L be the regular language accepted by the finite-state automaton of Figure 2.4.1.

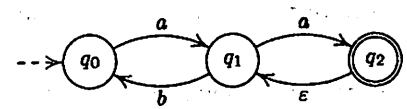


Fig. 2.4.1. A finite-state automaton

Using the terminology in the proof of the pumping lemma (Theorem 2.4.1), L has the constant $m = 3$.

On input $w = ababaa$, the finite-state automaton goes through the sequence $q_0, q_1, q_0, q_1, q_0, q_1, q_2$ of states. For such an input the pumping lemma provides the decomposition $x = \varepsilon$, $y = ab$, $z = abaa$; and the decomposition $x = a$, $y = ba$, $z = baa$. The first decomposition is due to the first repetition of state q_0; the second is a result of to the first repetition of state q_1.

For each string w of a minimum length 3, the pumping lemma implies a decomposition xyz in which the string y must be either ab or ba or ac. If $y = ab$, then $x = \varepsilon$ and the repetition of q_0 is assumed. If $y = ba$, then $x = a$ and the repetition of q_1 is assumed. If $y = ac$, then $x = a$ and the repetition of q_1 is assumed. □

Applications of the Pumping Lemma

For proving that a given language L is not regular, the pumping lemma implies the following schema of reduction to contradiction.

a. For the purpose of the proof assume that L is a regular language.

b. Let m denote the constant implied by the pumping lemma for L, under the assumption in (a) that L is regular.

c. Find a string w in L, whose length is at least m. Require that w implies a k, for each decomposition xyz of w, such that xy^kz is not in L. That is, find a w that implies, by using the pumping lemma, that a string not in L must, in fact, be there.

d. Use the contradiction in (c) to conclude that the pumping lemma does not apply for L.

e. Use the conclusion in (d) to imply that the assumption in (a), that L is regular, is false.

It should be emphasized that in the previous schema the pumping lemma implies only the existence of a constant m for the assumed regular language L, and the existence of a decomposition xyz for the chosen string w. This lemma does not provide any information about the specific values of m,x,y, and z besides the restriction that they satisfy the conditions $|xy| \leqslant m$ and $|y| > 0$. The importance for the schema of the condition $|xy| \leqslant m$ lies in allowing some limitation on the possible decompositions that are to be considered for the chosen w. The importance of the restriction $|y| > 0$ is in enabling a proper change in the pumped string.

Example 2.4.2 Consider the nonregular language $L = \{0^n1^n \mid n \geqslant 0\}$. To prove that L is nonregular assume to the contrary that it is regular. From the assumption that L is regular deduce the existence of a fixed constant m that satisfies the conditions of the pumping lemma for L.

Choose the string $w = 0^m1^m$ in L. By the pumping lemma, 0^m1^m has a decomposition of the form xyz, where $|xy| \leqslant m, |y| > 0$, and xy^kz is in L for each $k \geqslant 0$. That is, the decomposition must be of the form $x = 0^i, y = 0^j$, and $z = 0^{m-i-j}1^m$ for some i and j such that $j > 0$. (Note that the values of i,j, and m cannot be chosen arbitrarily.) Moreover, xy^0z must be in L. However, $xy^0z = 0^{m-j}1^m$ cannot be in L because $j > 0$. It follows that the pumping lemma does not apply for L, consequently contradicting the assumption that L is regular.

Other choices of w can also be used to show that L is not regular. However, they might result in a more complex analysis. For instance, for $w = 0^{m-1}1^{m-1}$ the pumping lemma provides three possible forms of decompositions:

a. $x = 0^i$, $y = 0^j$, $z = 0^{m-i-j-1}1^{m-1}$ for some $j > 0$.

b. $x = 0^{m-1-j}$, $y = 0^j1$, $z = 1^{m-2}$ for some $j > 0$.

c. $x = 0^{m-1}$, $y = 1$, $z = 1^{m-2}$.

In such a case, each of the three forms of decompositions must be shown to be inappropriate to conclude that the pumping lemma does not apply to w. For (a) the choice of $k = 0$ provides $xy^0z = 0^{m-1-j}1^{m-1}$ not in L. For (b) the choice of $k = 2$ provides $xy^2z = 0^{m-1}10^j1^{m-1}$ not in L. For (c) the choice of

$c = 0$ provides $xy^0z = 0^{m-1}1^{m-2}$ not in L. □

Example 2.4.3 Consider the nonregular language $L = \{\alpha\alpha^{rev} \mid \alpha \text{ is in } \{a,b\}^*\}$. To prove that L is not regular assume to the contrary that it is regular. Then deduce the existence of a fixed constant m that satisfies the conditions of the pumping lemma for L.

Choose $w = a^mbba^m$ in L. By the pumping lemma, $a^mbba^m = xyz$ for some x,y, and z such that $|xy| \leqslant m, |y| > 0$ and xy^kz is in L for each $k \geqslant 0$. That is, $x = a^i, y = a^j$, and $z = a^{m-i-j}bba^m$ for some i and j such that $j > 0$. However, $xy^0z = a^{m-j}bba^m$ is not in L, therefore contradicting the assumption that L is regular.

It should be noted that not every choice for w implies the desired contradiction. For instance, consider the choice of a^{2m} for w. By the pumping lemma, a^{2m} has a decomposition xyz in which $x = a^i, y = a^j$, and $z = a^{2m-i-j}$ for some i and j such that $j > 0$. With such a decomposition, $xy^kz = a^{2m+(k-1)j}$ is not in L if and only if $2m+(k-1)j$ is an odd integer. On the other hand, $2m+(k-1)j$ is an odd integer if and only if k is an even number and j is an odd number. However, although k can arbitrarily be chosen to equal any value, such is not the case with j. Consequently, the choice of a^{2m} for w does not guarantee the desired contradiction. □

A Generalization to the Pumping Lemma

The proof of the pumping lemma is based on the observation that a state is repeated in each computation on a "long" input, with a portion of the input being consumed between the repetition. The repetition of the state allows the pumping of the subcomputation between the repetition to obtain new accepting computations on different inputs. The proof of the pumping lemma with minor modifications also holds for the following more general theorem.

Theorem 2.4.2 For each relation R that is computable by a finite-state transducer, there exists a constant m that satisfies the following conditions. If (v,w) is in R and $|v|+|w| \geqslant m$, then v can be written as $x_vy_vz_v$ and w can be written as $x_wy_wz_w$, where $(x_vy_v^kz_v, x_wy_w^kz_w)$ is in R for each $k \geqslant 0$. Moreover, $|x_vy_v|+|x_wy_w| \leqslant m$, and $|y_v|+|y_w| > 0$.

A schema, similar to the one that uses the pumping lemma for determining nonregular languages, can utilize Theorem 2.4.2 for determining relations that are not computable by finite-state transducers.

Example 2.4.4 The relation $R = \{(u,u^{rev}) \mid u \text{ is in } \{0,1\}^*\}$ is not computable by a finite-state transducer. If R were computable by a finite-state transducer, then there would be a constant m that satisfies the conditions of Theorem 2.4.2 for R. In such a case, since $(0^m1^m, 1^m0^m)$ is in R, then $u = 0^m1^m$ could

be written as $x_v y_v z_v$ and $u^{rev} = 1^m 0^m$ could be written as $x_w y_w z_w$, where

$$x_v = 0^{i_v},\ y_v = 0^{j_v},\ z_v = 0^{m-i_v-j_v}1^m,$$
$$x_w = 1^{i_w},\ y_w = 1^{j_w},\ z_w = 1^{m-i_w-j_w}0^m,\ \textit{and}$$
$$j_v + j_w > 0.$$

Moreover, it would be implied that

$$(x_v y_v^0 z_v, x_w y_w^0 z_w) = (0^{m-j_v}1^m, 1^{m-j_w}0^m)$$

must also be in R, which is not the case. □

2.5 Closure Properties for Finite-Memory Programs

A helpful approach in simplifying the task of programming is to divide the given problem into subproblems, design subprograms to solve the subproblems, and then combine the subprograms into a program that solves the original problem. To allow for a similar approach in designing finite-state transducers (and finite-memory programs), it is useful to determine those operations that preserve the set of relations that are computable by finite-state transducers. Such knowledge can then be used in deciding how to decompose given problems to simpler subproblems, as well as in preparing tools for automating the combining of subprograms into programs.

In general, a set is said to be *closed* under a particular operation if each application of the operation on elements of the set results in an element of the set.

Example 2.5.1 The set of natural numbers is closed under addition, but it is not closed under subtraction. The set of integers is closed under addition and subtraction, but not under division. The set

$$\{S \mid S \text{ is a set of five or more integers}\}$$

is closed under union, but not under intersection or complementation. The set

$$\{S \mid S \text{ is a set of at most five integer numbers}\}$$

is closed under intersection, but not under union or complementation. □

The first theorem in this section is concerned with closure under the operation of union.

Theorem 2.5.1 The class of relations computable by finite-state transducers is closed under union.

Proof Consider any two finite-state transducers

$$M_1 = \langle Q_1, \Sigma_1, \Delta_1, \delta_1, q_{01}, F_1 \rangle,$$
$$\textit{and}$$
$$M_2 = \langle Q_2, \Sigma_2, \Delta_2, \delta_2, q_{02}, F_2 \rangle.$$

With no loss of generality assume that the sets Q_1 and Q_2 of states are mutually disjoint, and that neither of them contains q_0.

Let M_3 be the finite-state transducer $\langle Q_3, \Sigma_3, \Delta_3, \delta_3, q_0, F_3 \rangle$, where

$$Q_3 = Q_1 \cup Q_2 \cup \{q_0\},$$
$$\Sigma_3 = \Sigma_1 \cup \Sigma_2,$$
$$\delta_3 = \delta_1 \cup \delta_2 \cup \{(q_0, \varepsilon, q_{01}, \varepsilon), (q_0, \varepsilon, q_{02}, \varepsilon)\},\ \textit{and}$$
$$F_3 = F_1 \cup F_2$$

(see Figure 2.5.1).

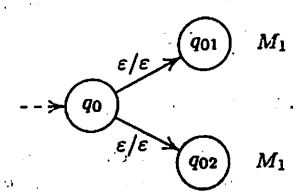


Fig. 2.5.1. A scheata of a finite-state transducer M_3 that computes $R(M_1) \cup R(M_2)$

Intuitively, M_3 is a finite-state transducer that at the start of each computation nondeterministically chooses to trace either a computation of M_1 or a computation of M_2.

By construction, $R(M_3) = R(M_1) \cup R(M_2)$. □

Besides their usefulness in simplifying the task of programming, closure properties can also be used to identify relations that cannot be computed by finite-state transducers.

Example 2.5.2 The union of the languages $L_1 = \{\varepsilon\}$ and $L_2 = \{0^i 1^i \mid i \geqslant 0\}$ is equal to the language $L_3 = \{0^i 1^i \mid i \geqslant 0\}$. By Theorem 2.5.1 the union $L_3 = L_1 \cup L_2$ of L_1 and L_2 is a regular language if L_1 and L_2 are regular languages. Since $L_1 = \{\varepsilon\}$ is a regular language, it follows that L_3 is a regular language if L_2 is a regular language. However, by Example 2.4.2 the language $L_3 = \{0^i 1^i \mid i \geqslant 0\}$ is not regular. Consequently, is also $L_2 = \{0^i 1^i \mid i \geqslant 1\}$ not regular. □

The relations $R_1 = \{(0^i 1^j, c^i) \mid i, j \geqslant 1\}$ and $R_2 = \{(0^i 1^j, c^j) \mid i, j \geqslant 1\}$ are computable by deterministic finite-state transducers. The pair $(0^i 1^j, c^k)$ is in R_1 if

and only if $k=i$, and it is in R_2 if and only if $k=j$. The intersection $R_1 \cap R_2$ contains all the pairs $(0^i1^j,c^k)$ that satisfy $k=i=j$, that is, $R_1 \cap R_2$ is the relation $\{(0^n1^n,c^n) \mid n \geq 1\}$.

If $R_1 \cap R_2$ is computable by a finite-state transducer then the language $\{0^n1^n \mid n \geq 1\}$ must be regular. However, by Example 2.4.2 the language is not regular. Therefore, the class of the relations that are computable by finite-state transducers is not closed under intersection.

The class of the relations computable by the finite-state transducers is also not closed under complementation. An assumption to the contrary would imply that the nonregular language $R_1 \cap R_2$ is regular, because by DeMorgan's law

$$R_1 \cap R_2 = \overline{(\overline{R_1} \cup \overline{R_2})}.$$

That is, an assumed closure under complementation would imply that $\overline{R_1}$ and $\overline{R_2}$ are computable by finite-state transducers. Theorem 2.5.1 would then imply that the union $\overline{R_1} \cup \overline{R_2}$ is computable by finite-state transducers. Finally, another application of the assumption would imply that

$$\overline{(\overline{R_1} \cup \overline{R_2})} = R_1 \cap R_2$$

is also computable by a finite-state transducer.

The choice of R_1 and R_2 also implies the nonclosure, under intersection, of the class of relations computable by deterministic finite-state transducers. The nonclosure under union and complementation, of the class of relations computable by deterministic finite-state transducers, is implied by the choice of the relations $\{(1,1)\}$ and $\{(1,11)\}$.

For regular languages the following theorem holds.

Theorem 2.5.2 Regular languages are closed under union, intersection, and complementation.

Proof By DeMorgan's law and the closure of regular languages under union (see Theorem 2.5.1), it is sufficient to show that regular languages are closed under complementation.

For the purpose of this proof consider any finite-state automaton

$$M = \langle Q, \Sigma, \delta, q_0, F \rangle.$$

By Theorem 2.3.1 it can be assumed that M is deterministic, and contains no ε transition rules.

Let M_{eof} be M with a newly added, nonaccepting "trap" state, say, q_{trap} and the following newly added transition rules.

a. (q,a,q_{trap}) for each pair (q,a) – of a state q in Q and of an input symbol a in Σ – for which no move is defined in M. That is, for each (q,a) for which no p exists in Q such that (q,a,p) is in δ.

b. (q_{trap},a,q_{trap}) for each input symbol a in Σ.

By construction M_{eof} is a deterministic finite-state automaton equivalent to M. Moreover, M_{eof} consumes all the inputs until their end, and it has no ε transition rules.

The complementation of the language $L(M)$ is accepted by the finite-state automaton $M_{complement}$ that is obtained from M_{eof} by interchanging the roles of the accepting and nonaccepting states.

For each given input $a_1...a_n$ the finite-state automaton $M_{complement}$ has a unique path that consumes $a_1...a_n$ until its end. The path corresponds to the sequence of moves that M_{eof} takes on such an input. Therefore, $M_{complement}$ reaches an accepting state on a given input if and only if M_{eof} does not reach an accepting state on the input. □

Example 2.5.3 Let M be the finite-state automaton whose transition diagram is given in Figure 2.5.2(a).

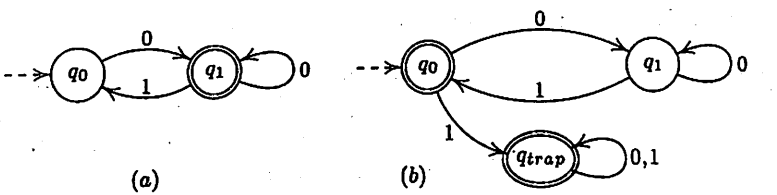


Fig. 2.5.2. The finite-state automaton in (b) accepts the compementation of the language that the finite-state automaton in (a) accepts

The complementation of $L(M)$ is accepted by the finite-state automaton whose transition diagram is given in Figure 2.5.2(b).

Without the trap state q_{trap}, neither M nor $M_{complement}$ would be able to accept the input 011, because none of them would be able to consume the whole input.

Without the requirement that the algorithm has to be applied only on deterministic finite-state automata, $M_{complement}$ could end up accepting an input that M also accepts. For instance, by adding the transition rule $(q_1,1,q_1)$ to M and $M_{complement}$, on input 01 each of the finite-state automata can end up either in state q_0 or in state q_1. In such a case, M would accept 01 because it can reach state q_1, and $M_{complement}$ would accept 01 because it can reach state q_0. □

2.6 Decidable Properties for Finite-Memory Programs

The emptiness problem, the equivalence problem, the halting problem, and other decision problems for finite-memory programs or, equivalently, for finite-state transducers are defined in a similar manner as for the general class of programs.

For instance, the equivalence problem for finite-state transducers asks for any given pair of finite-state transducers whether or not the transducers compute the same relation.

Similarly, the halting problem for finite-state transducers asks for any given pair (M,x), of a finite-state transducer M and of an input x for M, whether or not M has only halting computations on x.

In this section, some properties of finite-state transducers are shown to be decidable. The proofs are constructive in nature and they therefore imply effective algorithms for determining the properties in discourse. The first theorem is interesting mainly for its applications (see Example 2.1.2). It is concerned with the problem of determining whether an arbitrarily given finite-state automaton accepts no input.

Theorem 2.6.1 The emptiness problem is decidable for finite-state automata.

Proof Consider any finite-state automaton M. M accepts some input if and only if there is a path in its transition diagram from the node that corresponds to the initial state to a node that corresponds to an accepting state. The existence of such a path can be determined by the following algorithm.

Step 1 Mark in the transition diagram the node that corresponds to the initial state of M.

Step 2 Repeatedly mark those unmarked nodes in the transition diagram that are reachable by an edge from a marked node. Terminate the process when no additional nodes can be marked.

Step 3 If the transition diagram contains a marked node that corresponds to an accepting state, then determine that $L(M)$ is not empty. Otherwise, determine that $L(M)$ is empty. □

By definition, a program has only halting computations on inputs that it accepts. On the other hand, on each input that it does not accept, the program may have some computations that never terminate.

An important general determination about programs is whether they halt on all inputs. The proof of the following theorem indicates how, in the case of finite-memory programs, the uniform halting problem can be reduced to the

emptiness problem.

Theorem 2.6.2 The uniform halting problem is decidable for finite-state automata.

Proof Consider any finite-state automaton $M=\langle Q,\Sigma,\delta,q_0,F\rangle$. With no loss of generality, assume that the symbol c is not in Σ, and that Q has n states. In addition, assume that every state from which M can reach an accepting state by reading nothing is also an accepting state. Let A be a finite-state automaton obtained from M by replacing each ε transition rule of the form (q,ε,p) with a transition rule of the form (q,c,p). Let B be a finite-state automaton that accepts the language

$$\{x \mid x \text{ is in } (\Sigma\cup\{c\})^*, \text{ and } c^n \text{ is a substring of } x\}.$$

M has a nonhalting computation on a given input if and only if the following two conditions hold.

a. The input is not accepted by M.

b. On the given input M can reach a state that can be repeated without reading any input symbol.

Consequently, M has a nonhalting computation if and only if A accepts some input that has c^n as a substring.

By the proof of Theorem 2.5.2, a finite-state automaton C can be constructed to accept the complementation of $L(A)$. By that same proof, a finite-state automaton D can also be constructed to accept the intersection of $L(B)$ and $L(C)$.

By construction, D is a finite-state automaton that accepts exactly those inputs that have c^n as a substring and that are not accepted by A. That is, D accepts no input if and only if M halts on all inputs. The theorem thus follows from Theorem 2.6.1. □

For finite-memory programs that need not halt on all inputs, the proof of the following result implies an algorithm to decide whether or not they halt on specifically given inputs.

Theorem 2.6.3 The halting problem is decidable for finite-state automata.

Proof Consider any finite-state automaton M and any input $a_1...a_n$ for M. As in the proof of Theorem 2.3.1, one can derive for each $i=1,...,n$ the set $A_{a_1\cdots a_i}$ of all the states that can be reached by consuming $a_1...a_i$. Then M is determined to halt on $a_1...a_n$ if and only if either of the following two conditions hold.

a. $A_{a_1\cdots a_n}$ contains an accepting state.

b. For no integer i such that $1 \leqslant i \leqslant n$ the set $A_{a_1 \ldots a_i}$ contains a state that can be reached from itself by a sequence of one or more moves on ε transition rules. □

There are many other properties that are decidable for finite-memory programs. This section concludes with the following theorem.

Theorem 2.6.4 The equivalence problem is decidable for finite-state automata.

Proof Two finite-state automata M_1 and M_2 are equivalent if and only if the relation

$$(L(M_1) \cap \overline{L(M_2)}) \cup (\overline{L(M_1)} \cap L(M_2)) = \varnothing$$

holds, where $\overline{L(M_i)}$ denotes the complementation of $L(M_i)$ for $i=1,2$. The result then follows from the proof of Theorem 2.5.2 and from Theorem 2.6.1. □

The result in Theorem 2.6.4 can be shown to hold also for deterministic finite-state transducers (see Corollary 3.6.1). However, for the general class of finite-state transducers the equivalence problem can be shown to be undecidable (see Corollary 4.7.1).

2.7 Exercises

2.2.1 Let P be a program with k instruction segments and a domain of variables of cardinality m. Determine an upper bound on the number of states of P, and an upper bound on the number of possible transitions between these states.

2.2.2 Determine the diagram representation of a finite-state transducer that models the computations of the program in Figure 2.E.1.

```
x:=?
do
    read y
until y≠x
do
    y:=y+x
    write y
or
    if eof then accept
    reject
until false
```

Figure 2.E.1

Assume that the domain of the variables is $\{0,1\}$, and that 0 is the initial value in the domain. Denote each node in the transition diagram with the corresponding state of the program.

2.2.3 For each of the following relations give a finite-state transducer that computes the relation.

a. $\{(x\#y, a^i b^j) \mid x$ and y are in $\{a,b\}^*$, $i=$(number of a's in x), and $j=$(number of b's in $y)\}$

b. $\{(x, c^i) \mid x$ is in $\{a,b\}^*$, and $i=$(number of appearances of the substring abb's in $x)\}$

c. $\{(x, c^i) \mid x$ is in $\{a,b\}^*$, and $i=$(number of appearances of the substring aba's in $x)\}$

d. $\{(1^i, 1^j) \mid i$ and j are natural numbers and $i \geqslant j\}$

e. $\{(x,a) \mid x$ is in $\{0,1\}^*$, a is in $\{0,1\}$, and a appears at least twice in the string $x\}$

f. $\{(xy, a^i b^j) \mid x$ and y are in $\{a,b\}^*$, $i=$(the number of a's in x), and $j=$(the number of b's in $y)\}$

g. $\{(x,y) \mid x$ and y are in $\{a,b\}^*$, and either x is a substring of y or y is a substring of $x\}$

h. $\{(x,y) \mid x$ is in $\{a,b\}^*$, y is a substring of x, and the first and last symbols in y are of distinct values $\}$

i. $\{(x,y) \mid x$ and y are in $\{a,b\}^*$, and the substring ab has the same number of appearances in x and $y\}$

j. $\{(1^i, 1^j) \mid i=2j$ or $i=3j\}$

k. $\{(1^i, 1^j) \mid i \neq 2j\}$

l. $\{(x,y) \mid x$ and y are in $\{a,b\}^*$, and the number of a's in x differs from the number of b's in $y\}$

m. $\{(x,y) \mid x$ and y are in $\{0,1\}^*$, and (the natural number represented by y) $=3$(the natural number represented by $x)\}$

n. $\left\{\left(\binom{x_1}{y_1} \ldots \binom{x_s}{y_s}, z_1 \ldots z_n\right) \mid x_1, \ldots, x_n, y_1, \ldots, y_n, z_1, \ldots, z_n$ are in $\{0,1\}$, and (the natural number represented by $x_1 \ldots x_n$) $-$ (the natural number represented by $y_1 \ldots y_n$) $=$ (the natural number represented by $z_1 \ldots z_n)\}$

2.2.4 Let $M = \langle Q, \Sigma, \Delta, \delta, q_0, F \rangle$ be the deterministic finite-state transducer whose transition diagram is given in Figure 2.E.2.

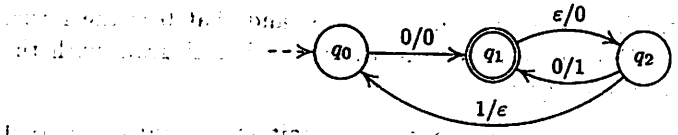


Figure 2.E.2

For each of the following relations find a finite-state transducer that computes the relation.

a. $\{(x,y) \mid x$ is in $L(M)$, and y is in $\Delta^*\}$.

b. $\{(x,y) \mid x$ is in $L(M)$, y is in Δ^*, and (x,y) is not in $R(M)\}$.

2.2.5 Show that if a deterministic finite-state transducer M accepts inputs x_1 and x_2 such that x_1 is a prefix of x_2, then on these inputs M outputs y_1 and y_2, respectively, such that y_1 is a prefix of y_2.

2.2.6 Determine the sequence of configurations in the computation that the finite-state transducer

$$\langle\{q_0,q_1,q_2\},\{0,1\},\{a,b\},\{(q_0,0,q_1,a),(q_1,1,q_0,a),(q_1,1,q_2,\varepsilon),(q_2,\varepsilon,q_1,b)\},q_0,\{q_2\}\rangle$$

has on input 0101.

2.2.7 Modify Example 2.2.16 for the case that M is the finite-state transducer whose transition diagram is given in Figure 2.2.2.

2.3.1 For each of the following languages construct a finite-state automaton that accepts the language:

a. $\{x \mid x$ is in $\{0,1\}^*$, and no two 0's are adjacent in $x\}$

b. $\{x \mid x$ is in $\{a,b,c\}^*$, and none of the adjacent symbols in x are equal $\}$

c. $\{x \mid x$ is in $\{0,1\}^*$, and each substring of length 3 in x contains at least two 1's$\}$

d. $\{1^z \mid z=3x+5y$ for some natural numbers x and $y\}$

e. $\{x \mid x$ is in $\{a,b\}^*$, and x contains an even number of a's and an even number of b's$\}$

f. $\{x \mid x$ is in $\{0,1\}^*$, and the number of 1's between every two 0's in x is even $\}$

g. $\{x \mid x$ is in $\{0,1\}^*$, and the number of 1's between every two substrings of the form 00 in x is even $\}$

h. $\{x \mid x$ is in $\{0,1\}^*$, but not in $\{10,01\}^*\}$

i. $\{x \mid x$ is in $\{a,b,c\}^*$, and a substring of x is accepted by the finite-state automaton of Figure 2.4.1 $\}$

2.3.2 Find a deterministic finite-state automaton that is equivalent to the finite-state automaton whose transition diagram is given in Figure 2.E.3.

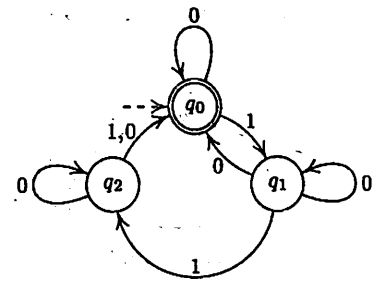


Figure 2.E.3

2.3.3 Find a Type 3 grammar that generates the language accepted by the finite-state automaton whose transition diagram is given in Figure 2.E.4.

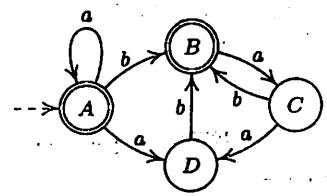


Figure 2.E.4

2.3.4 Find a finite-state automaton that accepts the language $L(G)$, for the case that $G=\langle N,\Sigma,P,S\rangle$ is the Type 3 grammar whose production rules are listed below.

$$\begin{aligned} S &\to aA \\ &\to bB \\ &\to b \\ A &\to aS \\ &\to bC \\ B &\to bS \\ &\to aC \\ &\to bA \\ C &\to aB \end{aligned}$$

2.3.5 Show that a language is generated by a Type 3 grammar if and only if it is generated by a right-linear grammar, and if and only if it is generated by a left-linear grammar.

2.3.6 Prove that a set is regular if and only if it is accepted by a finite-state automaton.

2.4.1 Let M be the finite-state automaton whose transition diagram is given in Figure 2.E.5.

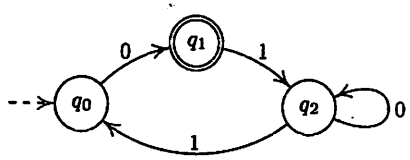


Figure 2.E.5

Using the notation of the proof of the pumping lemma for regular languages (Theorem 2.4.1), what are the possible values of m, x, and y for each w in $L(M)$?

2.4.2 Use the pumping lemma for regular languages to show that none of the following languages is regular.

a. $\{a^n b^t \mid n > t\}$

b. $\{v \mid v$ is in $\{a,b\}^*$, and v has fewer a's than b's$\}$

c. $\{x \mid x$ is in $\{a,b\}^*$, and $x = x^{rev}\}$

d. $\{vv^{rev} \mid v$ is accepted by the finite-state automaton of Figure 2.E.6$\}$

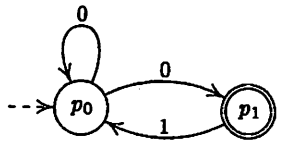


Figure 2.E.6

e. $\{a^{n^2} \mid n \geqslant 1\}$

f. $\{a^n b^t \mid n \neq t\}$

g. $\{x \mid x$ is in $\{a,b\}^*$, and $x \neq x^{rev}\}$

2.4.3 Show that each relation R computable by a finite-state transducer has a fixed integer m such that the following holds for all (v,w) in R. If $|w| > m \cdot \max(1,|v|)$, then $w = xyz$ for some x,y,z such that $(v,xy^k z)$ is in R for all $k \geqslant 0$. Moreover, $0 < |y| \leqslant m$.

2.4.4 Prove that the relation $\{(a^i b^j, c^k) \mid i$ and j are natural numbers and $k = i \cdot j\}$ is not computable by a finite-state transducer.

2.5.1 Let M_1 be the finite-state automaton given in Figure 2.E.3, and M_2 be the finite-state automaton given in Figure 2.E.6. Give a finite-state automaton that accepts the relation $R(M_1) \cap R(M_2)$.

2.5.2 For each of the following cases show that regular sets are closed under the operation Ψ.

a. $\Psi(L) = \{x \mid x$ is in L, and a proper prefix of x is in $L\}$.

b. $\Psi(L_1,L_2) = \{xyzw \mid xz$ is in L_1, and yw is in $L_2\}$.

2.5.3 Let Ψ be a permutation operation on languages defined as

$$\Psi(L) = \{x \mid x \text{ is a permutation of some } y \text{ in } L\}.$$

Show that regular sets are not closed under Ψ.

2.5.4 Show that the set of relations that finite-state transducers compute is closed under each of the following operations Ψ:

a. Inverse, that is, $\Psi(R) = R^{-1} = \{(y,x) \mid (x,y)$ is in $R\}$.

b. Closure, that is, $\Psi(R) = \cup_{i \geqslant 0} R^i$.

c. Composition, that is,

$$\Psi(R_1,R_2) = \{(x,y) \mid x = x_1x_2 \text{ and } y = y_1y_2 \text{ for some } (x_1,y_1) \text{ in } R_1, \text{ and some } (x_2,y_2) \text{ in } R_2\}.$$

d. Cascade composition, that is,

$$\Psi(R_1,R_2) = \{(x,z) \mid (x,y) \text{ is in } R_1 \text{ and } (y,z) \text{ is in } R_2 \text{ for some } y\}.$$

2.5.5 Show that the set of the relations computed by deterministic finite-state transducers is not closed under composition.

2.5.6 Let M be the finite-state automaton whose transition diagram is given in Figure 2.E.3. Give a finite-state automaton that accepts the complementation of $L(M)$.

2.5.7 Show that the complementation of a relation computable by a deterministic finite-state transducer, is computable by a finite-state transducer.

2.6.1 Show that the problem defined by the following pair is decidable:

Domain: $\{M \mid M$ is a finite-state automaton $\}$

Question: Is $L(M)$ a set of infinite cardinality for the given instance M ?

2.8 Bibliographic Notes

Finite-memory programs and their relationship to finite-state transducers have been studied in Jones and Muchnick (1977). Their applicability in designing

lexical analyzers can be seen in Aho, Sethi, and Ullman (1986). Their applicability in designing communication protocols is discussed in Danthine (1980). Their usefulness for solving systems of linear Diophantine equations follows from Büchi(1960). Finite-state transducers were introduced by Sheperdson (1959). Deterministic finite-state automata originated in McCulloch and Pitts (1943). Rabin and Scott (1959) introduced nondeterminism to finite-state automata, and showed the equivalency of nondeterministic finite-state automata to deterministic finite-state automata. The representation of finite-state transducers by transition diagrams is due to Myhill (1957). Chomsky and Miller (1958) showed the equivalency of the class of languages accepted by finite-state automata and the class of Type 3 languages. Kleene (1956) showed that the languages that finite-state automata accept are characterized by regular expressions. LEX is due to Lesk (1975). The pumping lemma for regular languages is due to Bar-Hillel, Perles, and Shamir (1961). Beauquier (see Ehrenfeucht, Parikh, and Rozenberg, 1981) showed the existence of a nonregular language that certifies the conditions of the pumping lemma. The decidability of the emptiness and equivalence problems for finite-state automata, as well as Exercise 2.6.1, have been shown by Moore (1956). Hopcroft and Ullman (1979) is a good source for additional coverage of these topics.

Chapter 3

RECURSIVE FINITE-DOMAIN PROGRAMS

Recursion is an important programming tool that deserves an investigation on its own merits. However, it takes on additional importance here by providing an intermediate class of programs – between the restricted class of finite-memory programs and the general class of programs. This intermediate class is obtained by introducing recursion into finite-domain programs. The first section of this chapter considers the notion of recursion in programs. The second section shows that recursive finite-domain programs are characterized by finite-state transducers that are augmented by pushdown memory. A grammatical characterization for the recursive finite-domain programs is provided in the third section. The fourth section considers the limitations of recursive finite-domain programs. And the fifth and sixth sections consider closure and decidable properties of recursive finite-domain programs, respectively.

3.1 Recursion

The task of programming is in many cases easier when recursion is allowed. However, although recursion does not in general increase the set of functions that the programs can compute, in the specific case of finite-domain programs such an increase is achieved.

Here recursion is introduced to programs by the instructions listed below.

a. Pseudoinstructions of the following form:

procedure ⟨*procedure name*⟩(⟨*list of formal parameters*⟩)
⟨*procedure body*⟩
end

These are used for defining procedures. Each list of formal parameters consists of variable names that are all distinct, and each procedure body consists of an arbitrary sequence of instructions.

b. Call instructions of the following form:

call ⟨*procedure name*⟩(⟨*list of actual parameters*⟩)

These are used for activating the execution of procedures. Each list of actual parameters is equal in size to the corresponding list of formal parameters, and it consists of variable names that are all distinct.

c. Return instructions of the following form:

return

These are used for deactivating the execution of procedures. The instructions are restricted to appearing only inside procedure bodies.

Finite-domain programs that allow recursion are called *recursive finite-domain programs.*

An execution of a call instruction activates the execution of the procedure that is invoked. The activation consists of copying the values from the variables in the list of actual parameters to the corresponding variables in the list of formal parameters, and of transferring the control to the first instruction in the body of the procedure.

An execution of a return instruction causes the deactivation of the last of those activations of the procedures that are still in effect. The deactivation causes the transfer of control to the instruction immediately following the call instruction that was responsible for this last activation. Upon the transfer of control, the values from the variables in the list of formal parameters are copied to the corresponding variables in the list of actual parameters. In addition, the variables that do not appear in the list of actual parameters are restored to their values just as before the call instruction was executed.

All the variables of a program are assumed to be recognized throughout the full scope of the program, and each of them is allowed to appear in an arbitrary number of lists of formal and actual variables.

Any attempt to enter or leave a procedure without using a call instruction or a return instruction, respectively, causes the program to abort execution in a rejecting configuration.

In what follows, each call instruction and each return instruction is considered to be an instruction segment.

Example 3.1.1 Let P be the recursive finite-domain program in Figure 3.1.1. The variables are assumed to have the domain {0,1}, with 0 as initial value. The program P accepts exactly those inputs in which the number of 0's is

equal to the number of 1's. On each such input the program outputs those input values that are preceded by the same number of 0's as 1's.

```
do                              /* I1 */
    if eof then accept          /* I2 */
    read x                      /* I3 */
    write x                     /* I4 */
    call RP(x)                  /* I5 */
until false                     /* I6 */

procedure RP(y)
    do                          /* I7 */
        read z                  /* I8 */
        if z≠y then             /* I9 */
            return              /* I10 */
        call RP(z)              /* I11 */
    until false                 /* I12 */
end
```

Figure 3.1.1 A recursive finite-domain program

On input 00111001 the program starts by reading the first input value 0 in I_3, writing 0 in I_4, and transferring the control to RP in I_5. Upon entering RP, $x=y=z=0$. In RP the program uses instruction segment I_8 to read the second input value 0, and then it calls RP recursively in I_{11}.

The embedded activation of RP reads the first 1 in the input and then executes the return instruction, to resume in I_{12} with $x=y=z=0$ the execution of the first activation of RP. The procedure continues by reading the second 1 of the input into z, and then returns to resume the execution of the main program in I_6 with $x=y=z=0$. The main program reads 1 into x, prints out that value, and invokes RP.

Upon entering RP, $x=y=1$ and $z=0$. The procedure reads 0 and then returns the control to the main program. The main program reads into x the last 0 of the input, prints the value out, and calls RP again. RP reads the last input value and returns the control to the main program, where the computation is terminated at I_2.

The table in Figure 3.1.2 shows the flow of data upon the activation and deactivation of RP. □

Values of x,y,z	Comments	Values of x,y,z	Comments
0 0 0	Initial values		
0 0 0	Call at I_5	1 0 0	Call at I_5
0 0 0	Continue at I_7	1 1 0	Continue at I_7
0 0 0	Call at I_{11}	1 1 0	Return
0 0 0	Continue at I_7	1 0 0	Continue at I_6
0 0 1	Return	0 0 0	Call at I_5
0 0 0	Continue at I_{12}	0 0 0	Continue at I_7
0 0 1	Return	0 0 1	Return
0 0 0	Continue at I_6	0 0 0	Continue at I_6

Fig. 3.1.2. Flow of data in the program of Figure 3.1.1 on input 00111001

```
call RP(parity)
if parity=0 then
    if eof then accept
reject

procedure RP(parity)
    do /* Process the next symbol in w. */
        read x
        write x
        parity:=1−parity
        call RP(parity)
    or /* Leave w and go to w^rev. */
        return
    until true
    /* Process the next symbol in w^rev. */
    read y
    if y≠x then reject
    return
end
```

Figure 3.1.3. A recursive finite-domain program

The definition given here for recursion is not standard, but can be shown to be equivalent to standard definitions. The sole motivation for choosing the nonstandard definition is because it simplifies the notion of states of recursive programs. The convention that the variables of a program are recognizable throughout the full scope of the program is introduced to allow uniformity in

the definition of states. The convention – that upon the execution of a return instruction the variables that do not appear in the list of actual parameters are restored to their values just before the execution of the corresponding call instructions – is introduced to show a resemblance to the notion of local variables in procedures.

Example 3.1.2 The recursive finite-domain program in Figure 3.1.3 computes the relation

$$\{(ww^{rev},w) \mid w \text{ is a string of even length in } \{0,1\}^*\}.$$

The domain of the variables is assumed to equal $\{0,1\}$, with 0 as initial value. On input 00111100 the program has a unique computation that gives the output 0011. The program makes five calls to the procedure *RP* while reading 0011. Then it proceeds with five returns while reading 1100. □

It turns out that an approach similar to the one used for studying finite-memory programs can also be used for studying recursive finite-domain programs. The main difference between the two cases is in the complexity of the argumentation.

Moreover, as in the case of finite-memory programs, it should be emphasized here that recursive finite-domain programs are important not only as a vehicle for investigating the general class of programs but also on their own merits. For instance, in many compilers the syntax analyzers are basically designed as recursive finite-domain programs. (The central task of a syntax analyzer is to group together, according to some grammatical rules, the tokens in the program that is compiled. Such a grouping enables the compiler to detect the structure of the program, and therefore to generate the object code.)

3.2 Pushdown Transducers

In general, recursion in programs is implemented by means of a pushdown store, that is, a last-in-first-out memory. Thus, it is only natural to suspect that recursion in finite-domain programs implicitly allows an access to some auxiliary memory. Moreover, the observation makes it also unsurprising that the computations of recursive finite-domain programs can be characterized by finite-state transducers that are augmented with a pushdown store. Such transducers are called pushdown transducers.

Pushdown Transducers

Each pushdown transducer M can be viewed as an abstract computing machine that consists of a *finite-state control*, an *input tape*, a read-only *input head*, a *pushdown tape* or *pushdown store*, a read-write *pushdown head*, an *output tape*,

and a write-only *output head* (see Figure 3.2.1). Each move of M is determined by the state of M, the input to be consumed, and the content on the top of the pushdown store. Each move of M consists of changing the state of M, reading at most one input symbol, changing the content on top of the pushdown store, and writing at most one symbol into the output.

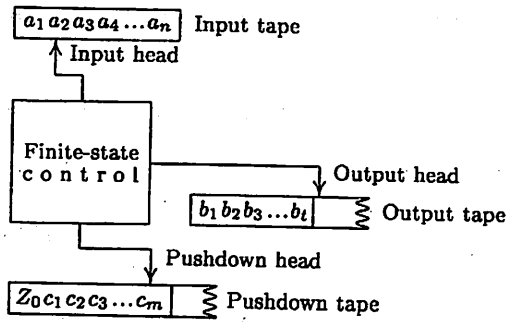
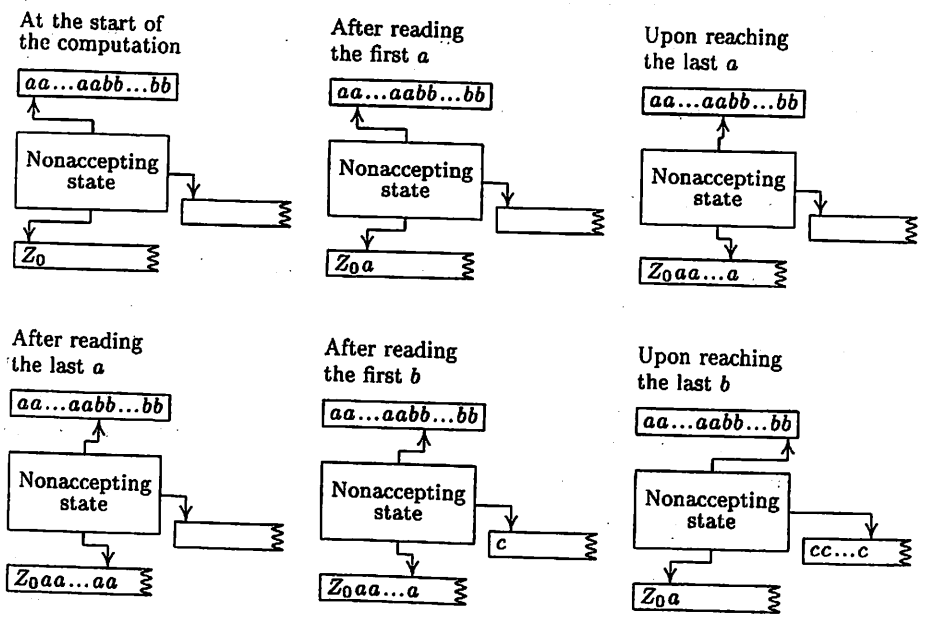


Fig. 3.2.1. Schemata of a pushdown transducer

Example 3.2.1 A pushdown transducer M can compute the relation $\{(a^i b^i, c^i) \mid i \geqslant 1\}$ by checking that each input has the form a... ab... b with the same number of a's as b's, and writing that many c's. The computations of M can be in the following manner (see Figure 3.2.2).



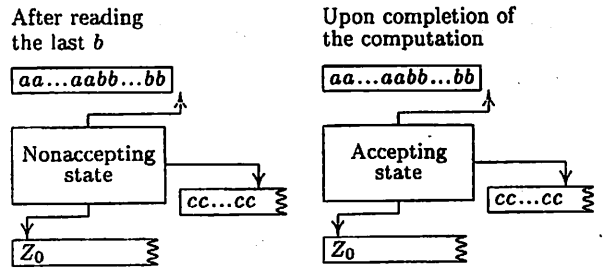


Fig. 3.2.2. A description of how a pushdown trasducer can compute the relation $\{(a^i b^i, c^i) \mid i \geqslant 1\}$

Initially the pushdown store is assumed to contain just one symbol, say, Z_0 to mark the bottom of the pushdown store. M starts each computation by reading the a's from the input tape while pushing them into the pushdown store. The symbols are read one at a time from the input.

Once M is done reading the a's from the input, it starts reading the b's. As M reads the b's it retrieves, or pops, one a from the pushdown store for each symbol b that it reads from the input. In addition, M writes one c to the output for each symbol b that it reads from the input.

M accepts the input if and only if it reaches the end of the input at the same time as it reaches the symbol Z_0 in the pushdown store. M rejects the input if it reaches the symbol Z_0 in the pushdown store before reaching the end of the input, because in such a case the input contains more b's than a's. M rejects the input if it reaches the end of the input before reaching the symbol Z_0 in the pushdown store, because in such a case the input contains more a's than b's. □

Formally, a mathematical system M consisting of an eight-tuple

$$\langle Q, \Sigma, \Gamma, \Delta, \delta, q_0, Z_0, F \rangle$$

is called a *pushdown transducer* if it satisfies the following conditions.

Q is a finite set, where the elements of Q are called the *states* of M.

Σ, Γ and Δ are alphabets. Σ is called the *input alphabet* of M, and its elements are called the *input symbols* of M. Γ is called the *pushdown alphabet* of M, and its elements are called the *pushdown symbols* of M. Δ is called the *output alphabet* of M, the elements of which are called the *output symbols* of M.

δ is a relation from $Q \times (\Sigma \cup \{\varepsilon\}) \times (\Gamma \cup \{\varepsilon\})$ to $Q \times \Gamma^* \times (\Delta \cup \{\varepsilon\})$. δ is called the *transition table* of M, the elements of which are called the *transition rules* of M.

q_0 is an element in Q, called the *initial state* of M.

Z_0 is an element in Γ, called the *bottom pushdown symbol* of M.

F is a subset of Q. The states in the subset F are called the *accepting*, or *final*, *states* of M.

In what follows, each transition rule $(q,\alpha,\beta,(p,\gamma,\rho))$ of a pushdown transducer will be written as $(q,\alpha,\beta,p,\gamma,\rho)$.

Example 3.2.2 $M=\langle Q,\Sigma,\Gamma,\Delta,\delta,q_0,Z_0,F\rangle$ is a pushdown transducer if

$Q=\{q_0,q_1,q_2\}$;
$\Sigma=\{a,b\}$;
$\Delta=\{a,b\}$;
$\Gamma=\{Z_0,c\}$;
$\delta=\{(q_0,a,\varepsilon,q_0,c,\varepsilon);(q_0,b,\varepsilon,q_0,c,\varepsilon),(q_0,\varepsilon,\varepsilon,q_1,\varepsilon,\varepsilon),(q_1,a,c,q_1,\varepsilon,a),$
$(q_1,b,c,q_1,\varepsilon,b),(q_1,\varepsilon,Z_0,q_2,Z_0,\varepsilon)\}$; *and*
$F=\{q_2\}$.

□

By definition, in each transition rule $(q,\alpha,\beta,p,\gamma,\rho)$ the entries q and p are states in Q, α is either an input symbol or an empty string, β is either a pushdown symbol or an empty string, γ is a string of pushdown symbols, and ρ is either an output symbol or an empty string.

Each pushdown transducer $M=\langle Q,\Sigma,\Gamma,\Delta,\delta,q_0,Z_0,F\rangle$ can be graphically represented by a *transition diagram* of the following form. For each state in Q the transition diagram has a corresponding node drawn as a circle. The initial state is identified by an arrow from nowhere that points to the corresponding node. Each accepting state is identified by a double circle. Each transition rule $(q,\alpha,\beta,p,\gamma,\rho)$ is represented by an edge from the node that corresponds to state q to the node that corresponds to state p. In addition, the edge is labeled with

$$\alpha/\beta/\gamma$$

For notational convenience, edges that agree in their origin and destination are merged, and their labels are separated by commas.

Example 3.2.3 Figure 3.2.3 gives the transition diagram for the pushdown transducer of Example 3.2.2.

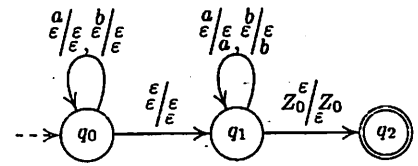


Fig. 3.2.3. A transition diagram of a pushdown transdcer

The label

$$\alpha/\varepsilon/\varepsilon$$

on the edge that starts and ends at state q_0 corresponds to the transition rule $(q_0,a,\varepsilon,q_0,\varepsilon,\varepsilon)$. The label

$$\varepsilon/\varepsilon/\varepsilon$$

on the edge that starts at state q_0 and ends at state q_1 corresponds to the transition rule $(q_0,\varepsilon,\varepsilon,q_1,\varepsilon,\varepsilon)$. □

The top row "$\alpha/$" in the label

$$\alpha/\beta/\gamma$$

corresponds to the input tape. The middle row "β/γ " corresponds to the pushdown tape. The bottom row "$/\rho$ " corresponds to the output tape.

Throughout the text the following conventions are assumed for each production rule $(q,\alpha,\beta,p,\gamma,\rho)$ of a pushdown transducer. The conventions do not affect the power of the pushdown transducers, and they are introduced to simplify the investigation of the pushdown transducers.

a. If $\beta=Z_0$, then Z_0 is a prefix of γ.

b. γ is a string of length 2 at most.

c. If γ is a string of length 2, the β is equal to the first symbol in γ.

Configurations and Moves of Pushdown Transducers

On each input x from Σ^* the pushdown transducer M has some set of possible configurations (see Figure 3.2.4). Each *configuration*, or *instantaneous description*, of M is a triplet (uqv,z,w), where q is a state of M, $uv=x$ is the input of M, z is a string from Γ^* of pushdown symbols, and w is a string from Δ^* of output symbols. Intuitively, a configuration (uqv,z,w) says that M on input x can reach state q with z in its pushdown store, after reading u and writing w. With no loss of generality it is assumed that Σ and Q are mutually disjoint.

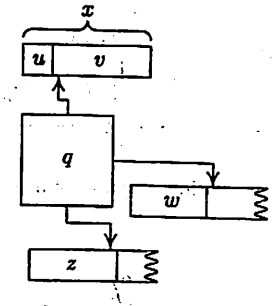


Fig. 3.2.4. A confuguration of a pushdown transducer

The configuration is said to be an *initial configuration* if $q=q_0$, $u=w=\varepsilon$, and $z=Z_0$. Such an initial configuration says that M is in its initial state q_0, with none of the input symbols being read yet (i.e., $u=\varepsilon$), with the output being still empty (i.e., $w=\varepsilon$), and the pushdown being still in its original stage (i.e., $z=Z_0$). In addition, the configuration says that M is given the input v.

The configuration is said to be an *accepting configuration* if $v=\varepsilon$ and q is an accepting state. Such an accepting configuration says that M reached an accepting state after reading all the input (i.e., $v=\varepsilon$) and writing w. In addition, the configuration says that the input M has consumed is equal to v.

Example 3.2.4 Consider the pushdown transducer M whose transition diagram is given in Figure 3.2.3. $(q_0abbb,Z_0,\varepsilon)$ is the initial configuration of M on input $abbb$. The configuration $(abq_1bb,Z_0cc,\varepsilon)$ of M says that M consumed already $u=ab$ from the input, the remainder of the input is $v=bb$, M has reached state q_1 with the string Z_0cc in the pushdown store, and the output so far is empty. The configurations are illustrated in Figure 3.2.5(a) and Figure 3.2.5(b), respectively.

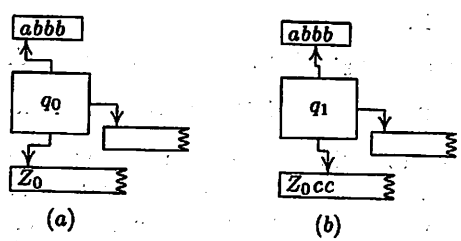


Fig. 3.2.5. Confugurations of the pushdown transducer of Figure 3.2.3

$(abbbq_2,Z_0,bb)$ and $(abq_2bb,Z_0cc,\varepsilon)$ are also configurations of M_0. The first configuration is accepting. The second, however, is not an accepting configuration despite its being in an accepting state, because the input has not been consumed until its end. □

The transition rules of M are used for defining the possible moves of M. Each move is in accordance with some transition rule. A *move* on transition rule

$(q,\alpha,\beta,p,\gamma,\rho)$ changes the state of the finite-state control from q to p; reads α from the input tape, moving the input head $|\alpha|$ positions to the right; writes ρ in the output tape, moving the output head $|\rho|$ positions to the right; and replaces on top of the pushdown store (i.e., from the location of the pushdown head to its left) the string β with the string γ, moving the pushdown head $|\gamma|-|\beta|$ positions to the right. The move is said to be a *pop* move if $|\gamma|<|\beta|$. The move is said to be a *push* move if $|\beta|<|\gamma|$. The symbol under the pushdown head is called the *top symbol* of the pushdown store.

A move of M from configuration C_1 to configuration C_2 is denoted $C_1\vdash_M C_2$, or simply $C_1\vdash C_2$ if M is understood. A sequence of zero or more moves of M from configuration C_1 to configuration C_2 is denoted $C_1\vdash^*_M C_2$, or simply $C_1\vdash^* C_2$, if M is understood.

Example 3.2.5 The pushdown transducer whose transition diagram is given in Figure 3.2.3, has a sequence of moves on input $abbb$ that is given by the following sequence of configurations:

$$(q_0abbb,Z_0,\varepsilon)\vdash(aq_0bbb,Z_0c,\varepsilon)\vdash(abq_0bb,Z_0cc,\varepsilon)\vdash(abq_1bb,Z_0cc,\varepsilon)\vdash$$
$$(abbq_1b,Z_0c,b)\vdash(abbbq_1,Z_0,bb)\vdash(abbbq_2,Z_0,bb).$$

This sequence is the only one that can start at the initial configuration and end at an accepting configuration for the input $abbb$. The sequence of configurations is depicted graphically in Figure 3.2.6.

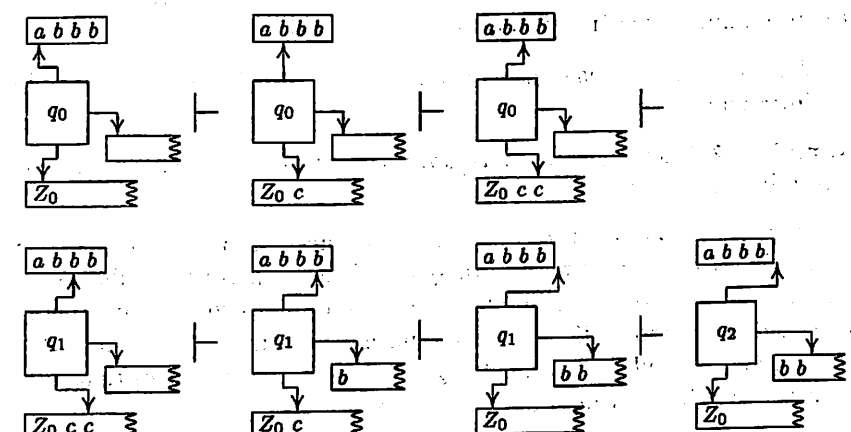


Fig. 3.2.6. Transition between configurations of the pushdown transducer of Figure 3.2.3

All the moves of M on the transition rules that both start and end at state q_0 are push moves. All the moves of M on the transition rules that both start and end at state q_1 are pop moves. □

A string in the pushdown store that starts at the bottom symbol and ends at the top symbol, excluding the bottom symbol, is called the *content* of the pushdown store. The pushdown store is said to be *empty* if its content is empty.

Example 3.2.6 Let M be the pushdown transducer of Figure 3.2.3. Consider the computation of M on input *abbb* (see Figure 3.2.6). M starts with an empty pushdown store, adding c to the store during the first move. After the second move, the content of the pushdown store is cc. The content of the pushdown store does not change during the third move. □

Determinism and Nondeterminism in Pushdown Transducers

The definitions of determinism and nondeterminism in pushdown transducers are, in principal, similar to those provided for finite-state transducers. The difference arises only in the details.

A pushdown transducer $M = \langle Q, \Sigma, \Gamma, \Delta, \delta, q_0, Z_0, F \rangle$ is said to be *deterministic* if for each state q in Q; each input symbol a in Σ; and each pushdown symbol Z in Γ, the union

$$\delta(q,a,Z) \cup \delta(q,a,\varepsilon) \cup \delta(q,\varepsilon,Z) \cup \delta(q,\varepsilon,\varepsilon),$$

is a multiset that contains at most one element.

Intuitively, M is deterministic if the state and the top pushdown symbol are sufficient for determining whether or not a symbol is to be read from the input, and the state, the top pushdown symbol, and the input to be read are sufficient for determining which transition rule is to be used.

A pushdown transducer is said to be *nondeterministic* if it is not a deterministic pushdown transducer.

Example 3.2.7 Let M_1 be the pushdown transducer whose transition diagram is given in Figure 3.2.3.

In a move from state q_1, the pushdown transducer M_1 reads an input symbol if and only if the topmost pushdown symbol is not Z_0. If the symbol is not Z_0, then the next symbol in the input uniquely determines which transition rule is to be used in the move. If the topmost pushdown symbol is Z_0, then M_1 must use the transition rule that leads to q_2. Consequently, the moves that originate at state q_1 can be fully determined "locally."

On the other hand, the moves from state q_0 cannot be determined locally, because the topmost pushdown symbol is not sufficient for determining if an input symbol is to be read in the move.

It follows that M_1 is a nondeterministic pushdown transducer. However, the pushdown transducer M_2 whose transition diagram is given in Figure 3.2.7 is

deterministic.

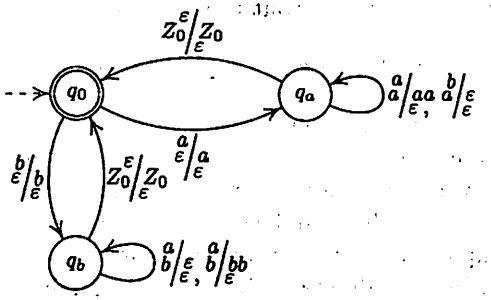


Fig. 3.2.7. A deterministic pushdown transducer

To move from state q_0 the pushdown transducer M_2 has to read an input symbol. If it reads the symbol a, then the move takes M_2 to state q_a. If it reads the symbol b, then the move takes M_2 to state q_b.

The topmost symbol in the pushdown store determines whether M_2 must enter state q_0 or state q_a on a move that originates at state q_a. If the topmost symbol is Z_0, then M moves to state q_0. If the topmost symbol is a, then M moves to state q_a. In the latter case M uses the transition rule (q_a,a,a,q_a,aa,c) if the input symbol to be read is a, and it uses the transition rule $(q_a,b,a,q_a,\varepsilon,\varepsilon)$ if the symbol to be read is b. □

Computations of Pushdown Transducers

The computations of the pushdown transducers are also defined like the computations of the finite-state transducers. An *accepting computation* of a pushdown transducer M is a sequence of moves of M that starts at an initial configuration and ends at an accepting one. A *nonaccepting*, or *rejecting*, *computation* of M is a sequence of moves on an input x, for which the following conditions hold.

a. The sequence starts from the initial configuration of M on input x.

b. If the sequence is finite, it ends at a configuration from which no move is possible.

c. M has no accepting computation on input x.

Each accepting computation and each nonaccepting computation of M is said to be a *computation* of M.

A computation is said to be a *halting computation* if it consists of a finite number of moves.

Example 3.2.8 Consider the pushdown transducer M whose transition diagram is given in Figure 3.2.7. The pushdown transducer has accepting computations only on those inputs that have the same number of a's as b's. On each

input w in which the pushdown transducer has an accepting computation, it writes the string c^i onto the output tape, where

$$i = (\text{the number of } a\text{'s in } w) = (\text{the number of } b\text{'s in } w).$$

The pushdown transducer enters state q_0 whenever the portion of the input read so far contains the same number of a's and b's. The pushdown transducer enters state q_a whenever the portion of the input read so far contains more a's than b's. Similarly, the pushdown transducer enters state q_b whenever the portion of the input read so far contains more b's than a's. The pushdown store is used for recording the difference between the number of a's and the number of b's, at any given instant of a computation.

On input *aabbba* the pushdown transducer M has only one computation. M starts the computation by moving from state q_0 to state q_a, while reading a, writing c, and pushing a into the pushdown store. In the second move M reads a, writes c, pushes a into the pushdown store, and goes back to q_a. In the third and fourth moves M reads b, pops a from the pushdown store, and goes back to state q_a. In the fifth move M goes to state q_0 without reading, writing, or changing the content of the pushdown store. In the sixth move M reads b, pushes b into the pushdown store, and moves to state q_b. In its seventh move M reads a, pops b from the pushdown store, writes c, and goes back to q_b. The computation terminates in an accepting configuration by a move from state q_b to state q_0 in which no input is read, no output is written, and no change is made in the content of the pushdown store. □

By definition, each move in each computation must be on a transition rule that keeps the computation in a path, that eventually causes the computation to read all the input and halt in an accepting state. Whenever more than one such alternative in the set of feasible transition rules exists, then any of these alternatives can be chosen. Similarly, whenever none of the feasible transition rules satisfies the conditions above, then any of these transition rules can be chosen. This observation suggests that we view the computations of the pushdown transducers as also being executed by imaginary agents with magical power.

An input x is said to be *accepted*, or *recognized*, by a pushdown transducer M if M has an accepting computation on x. An accepting computation on x that terminates in a configuration of the form (xq_f,z,w) is said to have an *output* w. The output of a nonaccepting computation is assumed to be undefined.

Example 3.2.9 Consider the pushdown transducer M, whose transition diagram is given in Figure 3.2.3. The pushdown transducer accepts exactly those inputs that have even length. In each accepting computation the pushdown transducer outputs the second half of the input.

As long as the pushdown transducer is in state q_0, it repeatedly reads an input symbol and stores c in the pushdown store. Alternatively, as long as the pushdown transducer is in state q_1, it repeatedly reads an input symbol and pops c from the pushdown store.

Upon reaching an empty pushdown store, the pushdown transducer makes a transition from state q_1 to state q_2 to verify that the end of the input has been reached. Consequently, in its accepting computations, the pushdown transducer must make a transition from state q_0 to state q_1 upon reaching the middle of its inputs.

On input *abbb* the pushdown transducer starts (its computation) with two moves, reading the first two input symbols, pushing two c's into the pushdown store, and returning to state q_0. In its third move the pushdown transducer makes a transition from state q_0 to state q_1.

The pushdown transducer continues with two moves, reading the last two symbols in the input, popping two c's from the pushdown store, and copying the input being read onto the output tape.

The pushdown concludes its computation on input *abbb* by moving from state q_1 to state q_2.

If M on input *abbb* reads more than two input symbols in the moves that originate at state q_0, it halts in state q_1 because of an excess of symbols in the pushdown store. If M on input *abbb* reads fewer than two input symbols in the moves that originates at state q_1, it halts in state q_1 because of a lack of symbols in the pushdown store. In either case the sequences of moves do not define computations of M. □

This example shows that, on inputs accepted by a pushdown transducer, the transducer may also have executable sequences of transition rules which are not considered to be computations.

Other definitions, such as those of the relations computable by pushdown transducers, the languages accepted by pushdown transducers, and the languages decided by pushdown transducers, are similar to those given for finite-state transducers in Section 2.2.

Example 3.2.10 The pushdown transducer M_1, whose transition diagram is given in Figure 3.2.3, computes the relation

$$\{(xy,y) \mid xy \text{ is in } \{a,b\}^*, \text{ and } |x| = |y|\}.$$

The pushdown transducer M_2, whose transition diagram is given in Figure

3.2.7 computes the relation

$\{(x,c^i) \mid x \text{ is in } \{a,b\}^*, \text{ and } i = (\text{number of } a\text{'s in } x) = (\text{number of } b\text{'s in } x)\}$.

□

From Recursive Finite-Domain Programs to Pushdown Transducers

The simulation of recursive finite-domain programs by pushdown transducers is similar to the simulation of the finite-memory programs by the finite-state transducers, as long as no call and return instructions are encountered. In such a case the pushdown transducers just trace across the states of the programs without using the pushdown store.

Upon reaching the call instructions, the pushdown transducers use their store to record the states from which the calls originate. Upon reaching the return instructions, the pushdown transducers retrieve from the store the states that activated the corresponding calls, and use this information to simulate the return instructions.

Specifically, consider any recursive finite-domain program P. Assume that P has m variables $x_1,...,x_m$, and k instruction segments $I_1,...,I_k$. Denote the initial value with $\odot$ in the domain of the variables of P. Let a *state* of P be an $(m+1)$-tuple $[i,v_1,...,v_m]$, where i is a positive integer no greater than k and $v_1,...,v_m$ are values from the domain of the variables of P.

The computational behavior of P can be modeled by a pushdown transducer $M = \langle Q,\Sigma,\Gamma,\Delta,\delta,q_0,Z_0,F\rangle$ whose states are used for recording the states of P, whose transition rules are used for simulating the transitions between the states of P, and whose pushdown store is used for recording the states of P which activated those executions of the procedures that have not been deactivated yet. $Q,\Sigma,\Gamma,\Delta,\delta,q_0,Z_0$, and F are defined in the following manner.

Q is a set containing of all those states that P can reach.

Σ is a set consisting of all those input values that P can read.

Γ is a set containing Z_0 and all the call states in Q. Z_0 is assumed to be a new element not in Q, and a call state is assumed to be a state that corresponds to a call instruction.

Δ is a set containing all the output values that P can write.

q_0 denotes the state $[1,\odot,...,\odot]$ of P.

F denotes the set of all those states in Q corresponding to an instruction of the form **if** *eof* **then accept.**

δ contains a transition rule of the form $(q,\alpha,\beta,p,\gamma,\rho)$ if and only if $q = [i,u_1,...,u_m]$ and $p = [j,v_1,...,v_m]$ are states in Q that satisfy the following conditions.

a. By executing the instruction segment I_i, the program P (with values $u_1,...,u_m$ in its variables $x_1,...,x_m$, respectively) can read α, write ρ, and reach instruction segment I_j with respective values $v_1,...,v_m$ in its variables.

b. If I_i is neither a call instruction nor a return instruction, then $\beta = \gamma = \varepsilon$. That is, the pushdown store is ignored.

c. If I_i is a call instruction, then $\beta = \varepsilon$ and $\gamma = q$. That is, the state q of P prior to invoking the procedure is pushed on top of the store. The state is recorded to allow the simulation of a return instruction that deactivates the procedure's activation caused by I_i.

d. If I_i is a return, instruction then β is assumed to be a state of P, and the transition from state q to state p is assumed to deactivate a call made at state β. In such a case $\gamma = \varepsilon$.

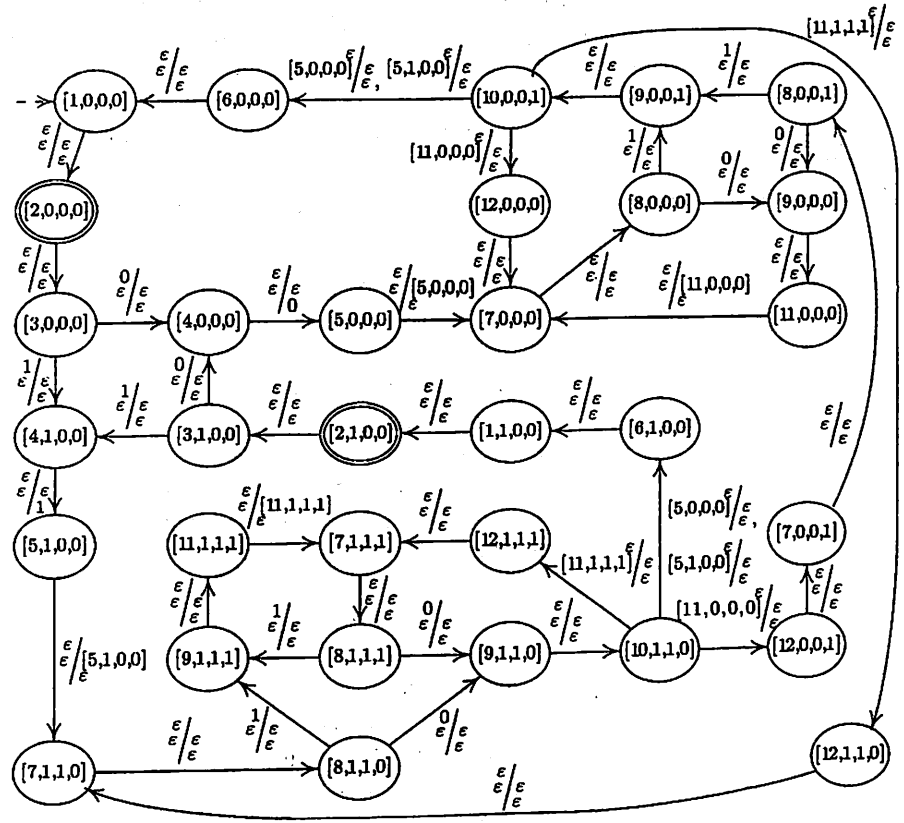


Fig. 3.2.8 Transition for a pushdown transducer that characterizes the recursive finite-domain program of Figure 3.1.1

Example 3.2.11 Consider the recursive finite-domain program P in Figure 3.1.1 with $\{0,1\}$ as the domain of its variables. The program is abstracted by the pushdown transducer whose transition diagram is given in Figure 3.2.8.

In the transition diagram, a state $[i,x,y,z]$ corresponds to instruction segment I_i with values x, y, and z in the variables x, y, and z, respectively.

On moving from state [3,0,0,0] to state [4,0,0,0], the pushdown transducer reads the value 0 into x. On moving from state [3,0,0,0] to state [4,1,0,0], the pushdown transducer reads the value 1 into x.

Each move from state [5,1,0,0] to state [7,1,1,0] corresponds to a call instruction, and each such move stores the state [5,1,0,0] in the pushdown store. In each such move, the value of y in state [7,1,1,0] is determined by the value of x in state [5,1,0,0], and the values of x and z in [7,1,1,0] are determined by the values of x and z in state [5,1,0,0].

Each move from state [10,1,1,0] to state [6,1,0,0] that uses the transition rule $([10,1,1,0],\varepsilon,[5,1,0,0],[6,1,0,0],\varepsilon,\varepsilon,\varepsilon)$ corresponds to an execution of a return instruction for a call that has been originated in state [5,1,0,0]. The value of x in state [6,1,0,0] is determined by the value of y in state [10,1,1,0]. The values of y and z in state [6,1,0,0] are determined by values of y and z in state [5,1,0,0].

The pushdown transducer has the following computation on input 0011.

$$
\begin{aligned}
([1,0,0,0]0011,Z_0,\varepsilon) &\vdash ([2,0,0,0]0011,Z_0,\varepsilon)\\
&\vdash ([3,0,0,0]0011,Z_0,\varepsilon)\\
&\vdash (0[5,0,0,0]011,Z_0,0)\\
&\vdash (0[7,0,0,0]011,Z_0[5,0,0,0],0)\\
&\vdash (0[8,0,0,0]011,Z_0[5,0,0,0],0)\\
&\vdash (00[9,0,0,0]11,Z_0[5,0,0,0],0)\\
&\vdash (00[11,0,0,0]11,Z_0[5,0,0,0],0)\\
&\vdash (00[7,0,0,0]11,Z_0[5,0,0,0][11,0,0,0],0)\\
&\vdash (00[8,0,0,0]11,Z_0[5,0,0,0][11,0,0,0],0)\\
&\vdash (001[9,0,0,1]1,Z_0[5,0,0,0][11,0,0,0],0)\\
&\vdash (001[10,0,0,1]1,Z_0[5,0,0,0][11,0,0,0],0)\\
&\vdash (001[12,0,0,0]1,Z_0[5,0,0,0],0)\\
&\vdash (001[7,0,0,0]1,Z_0[5,0,0,0],0)\\
&\vdash (001[8,0,0,0]1,Z_0[5,0,0,0],0)\\
&\vdash (0011[9,0,0,1],Z_0[5,0,0,0],0)\\
&\vdash (0011[10,0,0,1],Z_0[5,0,0,0],0)\\
&\vdash (0011[6,0,0,0],Z_0,0)\\
&\vdash (0011[1,0,0,0],Z_0,0)
\end{aligned}
$$

□

From Pushdown Transducers to Recursive Finite-Domain Programs

Using the previous discussion, we conclude that there is an algorithm that translates any given recursive finite-domain program into an equivalent pushdown transducer. Conversely, there is also an algorithm that derives an equivalent recursive finite-domain program from any given pushdown transducer $M=\langle Q,\Sigma,\Gamma,\Delta,\delta,q_0,Z_0,F\rangle$. The recursive finite-domain program can be a table-driven program of the form shown in Figure 3.2.9. The program simulates the pushdown transducer in a manner similar to that of simulating a finite-state transducer by a finite-memory program as shown in Section 2.2. The main difference is in simulating the effect of the pushdown store.

```
state := q0
do
    top := Z0
    call RP(top) /* Record the bottom pushdown symbol Z0. */
until false

procedure RP(top)
    do
        /* Accept if an accepting state of M is reached at the end
        of the input. */
        if F(state) then
            if eof then accept
        /* Nondeterministically find the entries of the transition rule
        (q,α,β,p,γ,ρ) that M uses in the next simulated move. */
        do in := e or read in until true /* in := α */
        do pop := e or pop := top until true /* pop := β */
        next_state :=? /* next_state := p */
        push :=? /* push := γ */
        out :=? /* out := ρ */
        if not δ(state,in,pop,next_state,push,out) then reject
        /* Simulate the next move of M. */
        state := next_state
        if out ≠ e then write out
        if pop ≠ e then return
        if push ≠ e then call RP(push)
    until false
end
```

Figure 3.2.9 A table-driven recursive finite-domain program for simulating pushdown transducers.

The program uses the variable *state* for recording the states that M leaves in its moves, the variable *top* for recording the topmost symbol in the pushdown store,

the variable *in* for recording inputs that M consumes in its moves, the variable *next_state* for recording the states that M enters in its moves, the variable *pop* for recording the substrings that are replaced on top of the pushdown store, the variable *push* for recording the changes that have to be made on top of the pushdown store, and a variable *out* for recording the outputs that have to be written in the moves of M.

The content of the pushdown store is recorded indirectly through recursion. Each pushing of a symbol is simulated by a recursive call, and each popping of a symbol is simulated by a return.

The main program initializes the variable *state* to q_0, and calls *RP* to record a pushdown store containing only Z_0.

The body of the recursive procedure *RP* consists of an infinite loop. Each iteration of the loop starts by checking whether an accepting state of M has been reached at the end of the input. If such is the case, the program halts in an accepting configuration. Otherwise, the program simulates a single move of M. The predicate F is used to determine whether *state* holds an accepting state.

The simulation of each move of M is done in a nondeterministic manner. The program guesses the input to be read, the top portion of the pushdown store to be replaced, the state to be reached, the replacement to the top of the store, and the output to be written. Then the program uses the predicate δ for determining the appropriateness of the guessed values. The program aborts the simulation if it determines that the guesses are inappropriate. Otherwise, the program records the changes that have to be done as a result of the guessed transition rule.

The variables of the program are assumed to have the domain $Q \cup \Sigma \cup \Gamma \cup \Delta \cup \{e\}$, with e being a new symbol. In addition, with no loss of generality, it is assumed that each transition rule $(q,\alpha,\beta,p,\gamma,\rho)$ of M satisfies either $|\beta|+|\gamma|=1$ or $\beta=\gamma=Z_0$. The latter assumptions are made to avoid the situation in which both a removal and an addition of a symbol in the pushdown store are to be simulated for the same move of M.

Example 3.2.12 For the pushdown transducer of Figure 3.2.3 the table-driven program has the domain of variables equal to $\{a,b,Z_0,c,q_0,q_1,q_2,e\}$. The truth values of the predicates F and δ are defined by the corresponding tables in Figure 3.2.10. (F and δ are assumed to have the value `false` for arguments that are not specified in the tables.)

The pushdown transducer can be simulated also by the non-table-driven program of Figure 3.2.11. □

F	
q_0	false
q_1	false
q_2	true

δ	$q_0,\varepsilon,\varepsilon$	$q_1,\varepsilon,\varepsilon$	q_1,ε,a	q_1,ε,b	q_2,Z_0,ε
$q_0,\varepsilon,\varepsilon$	false	true	false	false	false
q_0,a,ε	true	false	false	false	false
q_0,b,ε	true	false	false	false	false
q_1,a,ε	false	false	true	false	false
q_1,b,ε	false	false	false	true	false
q_1,ε,Z_0	false	false	false	false	true

Fig. 3.2.10. Tables for a table-drven recursive finite-domain program that simulates the pushdown transducer of Figure 3.2.3

```
state := q0
next_top := Z0
call RP(next_top)

procedure RP(top)
    do
        if state = q0 then
        do
            read in
            if (in ≠ a) and (in ≠ b) then reject
            next_top := c
            call RP(next_top)
        or
            state := q1
        until true
        if state = q1 then
        do
            if top = Z0 then state := q2
            if top = c then
            do
                read in
                if (in ≠ a) and (in ≠ b) then reject
                write in
                return
            until true
        until false
        if state = q2 then
            if eof then accept
    until false
end
```

Figure 3.2.11 A non-table-driven recursive finite-domain program for simulating the pushdown transducer of Figure 3.2.3

In a manner similar to the one discussed in Section 2.2 for finite-state transducers, the recursive finite-domain program can be modified to be deterministic whenever the given pushdown transducer is deterministic.

A formalization of the previous discussion implies the following theorem.

Theorem 3.2.1 A relation is computable by a nondeterministic (respectively, deterministic) recursive finite-domain program if and only if it is computable by a nondeterministic (respectively, deterministic) pushdown transducer.

Pushdown Automata

Pushdown transducers whose output components are ignored are called pushdown automata. Formally, a *pushdown automaton* is a tuple $\langle Q,\Sigma,\Gamma,\delta,q_0,Z_0,F\rangle$, where Q,Σ,Γ,q_0,Z_0, and F are defined as for pushdown transducers, and δ is a relation from $Q\times(\Sigma\cup\{\varepsilon\})\times(\Gamma\cup\{\varepsilon\})$ to $Q\times\Gamma^*$.

As in the case for pushdown transducers, the following conditions are assumed for each transition rule $(q,\alpha,\beta,p,\gamma)$ of a pushdown automaton.

a. If $\beta=Z_0$, then Z_0 is a prefix of γ.

b. γ is a string of length 2 at most.

c. If γ is a string of length 2, then β is equal to the first symbol in γ.

Transition diagrams similar to those used for representing pushdown transducers can be used to represent pushdown automata. The only difference is that the labels of the edges do not contain entries for outputs.

Example 3.2.13 The pushdown automaton M that is induced by the pushdown transducer of Figure 3.2.3 is $\langle Q,\Sigma,\delta,q_0,F\rangle$, where

$$
\begin{aligned}
&Q=\{q_0,q_1,q_2\},\\
&\Sigma=\{a,b\},\\
&\Gamma=\{Z_0,c\},\\
&\delta=\{(q_0,a,\varepsilon,q_0,c),(q_0,b,\varepsilon,q_0,c),(q_0,\varepsilon,\varepsilon,q_1,\varepsilon),(q_1,a,c,q_1,\varepsilon),\\
&\quad (q_1,b,c,q_1,\varepsilon),(q_1,\varepsilon,Z_0,q_2,Z_0)\},\ \textit{and}\\
&F=\{q_2\}.
\end{aligned}
$$

The pushdown automaton is represented by the transition diagram of Figure 3.2.12. □

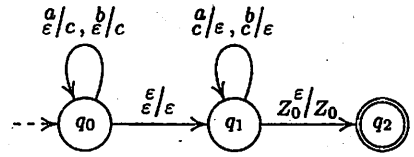


Fig. 3.2.12. A transtion diagram of a pushdown automaton

The pushdown automaton is said to be *deterministic* if for each state q in Q, each input symbol a in Σ, and each pushdown symbol Z in Γ the union

$$\delta(q,a,Z)\cup\delta(q,a,\varepsilon)\cup\delta(q,\varepsilon,Z)\cup\delta(q,\varepsilon,\varepsilon)$$

is a multiset that contains at most one element. The pushdown automaton is said to be *nondeterministic* if it is not a deterministic pushdown automaton.

A *configuration*, or an *instantaneous description*, of the pushdown automaton is a pair (uqv,z), where q is a state in Q, uv is a string in Σ^*, and z is a string in Γ^*. Other definitions, such as those for initial and final configurations, $\vdash_M$, $\vdash$, $\vdash_M^*$, $\vdash^*$; and acceptance, recognition, and decidability of a language by a pushdown automaton, are similar to those given for pushdown transducers.

Example 3.2.14 The transition diagram in Figure 3.2.13

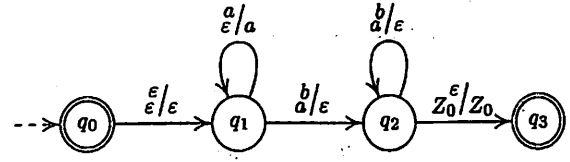


Fig. 3.2.13. Transtion diagram of a deterministic pushdown automaton that accepts $\{a^ib^i\mid i\geqslant 0\}$.

represents the deterministic pushdown automaton

$$
\begin{aligned}
&\langle\{q_0,q_1,q_2,q_3\},\{a,b\},\{a,Z_0\},\\
&\{(q_0,\varepsilon,\varepsilon,q_1,\varepsilon),(q_1,a,\varepsilon,q_1,a),(q_1,b,a,q_2,\varepsilon),(q_2,b,a,q_2,\varepsilon),(q_2,\varepsilon,Z_0,q_3,Z_0)\},\\
&q_0,Z_0,\{q_3\}\rangle.
\end{aligned}
$$

The pushdown automaton accepts the language $\{a^ib^i\mid i\geqslant 0\}$. The pushdown automaton reads the a's from the input and pushes them into the pushdown store as long as it is in state q_1. Then, it reads the b's from the input, while removing one a from the pushdown store for each b that is read. As long as it reads b's, the pushdown automaton stays in state q_2. The pushdown automaton enters the accepting state q_3 once it has read the same number of b's as a's.

The transition diagram in Figure 3.2.14

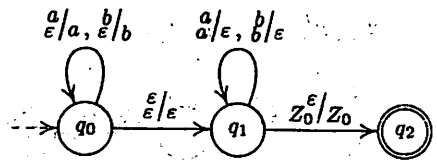


Fig. 3.2.14: Transtion diagram of a nondeterministic pushdown automaton that accepts $\{ww^{rev} \mid w \text{ is in } \{a,b\}^*\}$

is of a nondeterministic pushdown automaton that accepts the language $\{ww^{rev} \mid w \text{ is in } \{a,b\}^*\}$. In state q_0 the pushdown automaton reads w and records it in the pushdown store in reverse order. On the other hand, in state q_1 the pushdown automaton reads w^{rev} and compares it with the string recorded in the pushdown store. □

3.3 Context-Free Languages

Pushdown automata can be characterized by Type 2 grammars or, equivalently, by context-free grammars.

Specifically, a Type 0 grammar $G=\langle N,\Sigma,P,S\rangle$ is said to be *context-free* if each of its production rules has exactly one nonterminal symbol on its left hand side, that is, if each of its production rules is of the form $A \to \alpha$.

The grammar is called context-free because it provides no mechanism to restrict the usage of a production rule $A \to \alpha$ within some specific context. However, in a Type 0 grammar such a restriction can be achieved by using a production rule of the form $\beta A\gamma \to \beta\alpha\gamma$ to specify that $A \to \alpha$ is to be used only within the context of β and γ.

The languages that context-free grammars generate are called *context-free languages.*

Example 3.3.1 The language

$$\{a^{i_1}b^{i_1}a^{i_2}b^{i_2}...a^{i_n}b^{i_n} \mid n,i_1,...,i_n \geq 0\}$$

is generated by the context-free grammar $\langle N,\Sigma,P,S\rangle$, whose production rules are given below.

$$\begin{aligned} S &\to SS \\ &\to A \\ &\to \varepsilon \\ A &\to aAb \\ &\to ab \end{aligned}$$

□

From Context-Free Grammars to Type 2 Grammars

Recall that a Type 2 grammar is a context-free grammar $G=\langle N,\Sigma,P,S\rangle$ in which $A \to \varepsilon$ in P implies that $A=S$ and that no right-hand side of the production rules contains S. By the following theorem it follows that context-free grammars and Type 2 grammars act as "maximal" and "minimal" grammars for the same class of languages.

Theorem 3.3.1 Each context-free language is also a Type 2 language.

Proof Consider any context-free grammar $G_1=\langle N,\Sigma,P_1,S_1\rangle$. A Type 2 grammar $G_2=\langle N\cup\{S_2\},\Sigma,P_2,S_2\rangle$ satisfies $L(G_2)=L(G_1)$, if S_2 is a new symbol and P_2 is obtained from P_1 in the following way.

Initialize P_2 to equal $P_1\cup\{S_2 \to S_1\}$. Then, as long as P_2 contains a production rule of the form $A \to \varepsilon$ for some $A \neq S_2$, modify P_2 as follows.

a. Delete the production rule $A \to \varepsilon$ from P_2.

b. Add a production rule to P_2 of the form $B \to \alpha_A$ as long as such new production rules can be formed. α_A is assumed to be the string α with one appearance of A omitted in it, and α is assumed to be the right-hand side of a production rule of the form $B \to \alpha$ that is already in P_2. If $\alpha_A=\varepsilon$ and the production rule $B \to \varepsilon$ has been removed earlier from P_2, then the production rule is not reinserted to P_2.

No addition of a production rule of the form $B \to \alpha_A$ to P_2 changes the generated language, because any usage of the production rule can be simulated by the pair $B \to \alpha$ and $A \to \varepsilon$ of production rules.

Similarly, no deletion of a production rule $A \to \varepsilon$ from P_2 affects the generated language, because each subderivation

$$C \Rightarrow \beta_1 A\beta_2 \Rightarrow^* \gamma_1 A\gamma_2 \Rightarrow \gamma_1\gamma_2$$

which uses $A \to \varepsilon$ can be replaced with an equivalent subderivation of the form $C \Rightarrow \beta_1\beta_2 \Rightarrow^* \gamma_1\gamma_2$. □

Example 3.3.2 Let G_1 be the context-free grammar whose production rules are listed below.

$$\begin{aligned} S_1 &\to \varepsilon \\ &\to S_1aCC \\ C &\to \varepsilon \\ &\to Dab \\ D &\to S_1 \end{aligned}$$

The construction in the proof of Theorem 3.3.1 implies the following equivalent grammars, where G_2 is a Type 2 grammar.

G_1'	G_2'	G_3'	G_2
Added $S_2 \to S_1$	*Removed* $S_1 \to \varepsilon$	*Removed* $C \to \varepsilon$	*Removed* $D \to \varepsilon$
$S_2 \to S_1$	$S_2 \to S_1$	$S_2 \to S_1$	$S_2 \to S_1$
$S_1 \to \varepsilon$	$\to \varepsilon$	$\to \varepsilon$	$\to \varepsilon$
$\to S_1aCC$	$S_1 \to S_1aCC$	$S_1 \to S_1aCC$	$S_1 \to S_1aCC$
$C \to \varepsilon$	$\to aCC$	$\to S_1aC$	$\to S_1aC$
$\to Dab$	$C \to \varepsilon$	$\to S_1a$	$\to S_1a$
$D \to S_1$	$\to Dab$	$\to aCC$	$\to aCC$
	$D \to S_1$	$\to aC$	$\to aC$
	$\to \varepsilon$	$\to a$	$\to a$
		$C \to Dab$	$C \to Dab$
		$D \to S_1$	$\to ab$
		$\to \varepsilon$	$D \to S_1$

□

From Context-Free Grammars to Pushdown Automata

Pushdown automata and recursive finite-domain programs process their inputs from left to right. To enable such entities to trace derivations of context-free grammars, the following lemma considers a similar property in the derivations of context-free grammars.

Lemma 3.3.1 If a nonterminal symbol A derives a string ρ of terminal symbols in a context-free grammar G, then ρ has a leftmost derivation from A in G.

Proof The proof is by contradiction. Recall that in context-free grammars the leftmost derivations $\rho_1 \Rightarrow \rho_2 \Rightarrow ... \Rightarrow \rho_n$ replace the leftmost nonterminal symbol in each sentential form ρ_i, $i=1,2,...,n-1$.

The proof relies on the observation that the ordering in which the nonterminal symbols are replaced in the sentential forms is of no importance for the derivations in context-free grammars. Each nonterminal symbol in each sentential form is expanded without any correlation to its context in the sentential form.

Consider any context-free grammar G. For the purpose of the proof assume that a string ρ of terminal symbols has a derivation of length n from a nonterminal symbol A. In addition, assume that ρ has no leftmost derivation from A.

Let

$$A \Rightarrow \rho_1 \Rightarrow ... \Rightarrow \rho_m \Rightarrow ... \Rightarrow \rho_n = \rho$$

be a derivation of length n in which $A \Rightarrow \rho_1 \Rightarrow ... \Rightarrow \rho_m$ is a leftmost subderivation. In addition, assume that m is maximized over the derivations $A \Rightarrow^* \rho$ of length n. By the assumption that ρ has no leftmost derivation from A, it follows that $m < n-1$.

The derivation in question satisfies

$$\rho_m = wB\hat{\rho}_m,\ \rho_{m+1} = wB\hat{\rho}_{m+1},\ ...\ ,\ \rho_k = wB\hat{\rho}_k,\ \rho_{k+1} = w\beta\hat{\rho}_k$$

for some string w of terminal symbols, production rule $B \to \beta$, $m < k < n$, and $\hat{\rho}_m,...,\hat{\rho}_k$. Thus

$$A \Rightarrow \rho_1 \Rightarrow ... \Rightarrow \rho_{m-1} \Rightarrow \rho_m = wB\hat{\rho}_m \Rightarrow w\beta\hat{\rho}_m \Rightarrow$$
$$w\beta\hat{\rho}_{m+1} \Rightarrow ... \Rightarrow w\beta\hat{\rho}_k = \rho_{k+1} \Rightarrow ... \Rightarrow \rho_n = \rho$$

is also a derivation of ρ from A of length n.

However, in this new derivation

$$A \Rightarrow \rho_1 \Rightarrow ... \Rightarrow \rho_m \Rightarrow w\beta\hat{\rho}_m$$

is a leftmost subderivation of length $m+1$. Consequently, contradicting the existence of a maximal m as implied above, from the assumption that ρ has only nonleftmost derivations from A.

As a result, the assumption that ρ has no leftmost derivation from A is also contradicted. □

The proof of the following theorem shows how pushdown automata can trace the derivations of context-free grammars.

Theorem 3.3.2 Each context-free language is accepted by a pushdown automaton.

Proof Consider any context-free grammar $G = \langle N, \Sigma, P, S \rangle$. With no loss of generality assume that Z_0 is not in $N \cup \Sigma$. $L(G)$ is accepted by the pushdown automaton

$$M = \langle Q, \Sigma, N \cup \Sigma \cup \{Z_0\}, \delta, q_0, Z_0, \{q_f\} \rangle$$

whose transition table δ consists of the following derivation rules.

a. A transition rule of the form $(q_0, \varepsilon, \varepsilon, q_1, S)$.

b. A sequence of transition rules for each $A \to \alpha$ in P. Each such sequence starts and ends at state q_1, and replaces a nonterminal symbol A on top of the pushdown store with the string α in reverse order.

c. A transition rule of the form $(q_1, a, a, q_1, \varepsilon)$, for each terminal symbol a in the alphabet Σ.

d. A transition rule of the form $(q_1,\varepsilon,Z_0,q_f,Z_0)$.

Intuitively, we know that on a given input x the pushdown automaton M nondeterministically traces a leftmost derivation in G that starts at S and ends at x. At each stage of the tracing, the portion of the input that has already been read together with the content of the pushdown store in reverse order, record the sentential form in the corresponding stage of the derivation.

The transition rule in (a) is used for pushing the first sentential form S into the pushdown store. The transition rules in (b) are used for replacing the leftmost nonterminal symbol in a given sentential form with the right-hand side of an appropriate production rule. The transition rules in (c) are used for matching the leading terminal symbols in the sentential forms with the corresponding symbols in the given input x. The purpose of the production rule in (d) is to move the pushdown automaton into an accepting state upon reaching the end of a derivation.

By induction on n it can be verified that x has a leftmost derivation in G if and only if M has an accepting computation on x, where the derivation and the computation have the following forms with $u_iv_i=x$ for $1\leqslant i<n$.

$$
\begin{array}{ll}
S & (q_0x,Z_0)\vdash(q_1x,Z_0S)\\
\Rightarrow u_1A_1\rho_1 & \vdash^*(u_1q_1v_1,Z_0\rho_1^{rev}A_1)\\
\Rightarrow u_2A_2\rho_2 & \vdash^*(u_2q_1v_2,Z_0\rho_2^{rev}A_2)\\
\Rightarrow \ldots & \vdash^*\ldots\\
\Rightarrow u_{n-1}A_{n-1}\rho_{n-1} & \vdash^*(u_{n-1}q_1v_{n-1},Z_0\rho_{n-1}^{rev}A_{n-1})\\
\Rightarrow x & \vdash^*(xq_1,Z_0)\vdash(xq_f,Z_0)
\end{array}
$$

□

Example 3.3.3 If G is the context-free grammar of Example 3.3.1, then the language $L(G)$ is accepted by the pushdown automaton M, whose transition diagram is given in Figure 3.3.1(a).

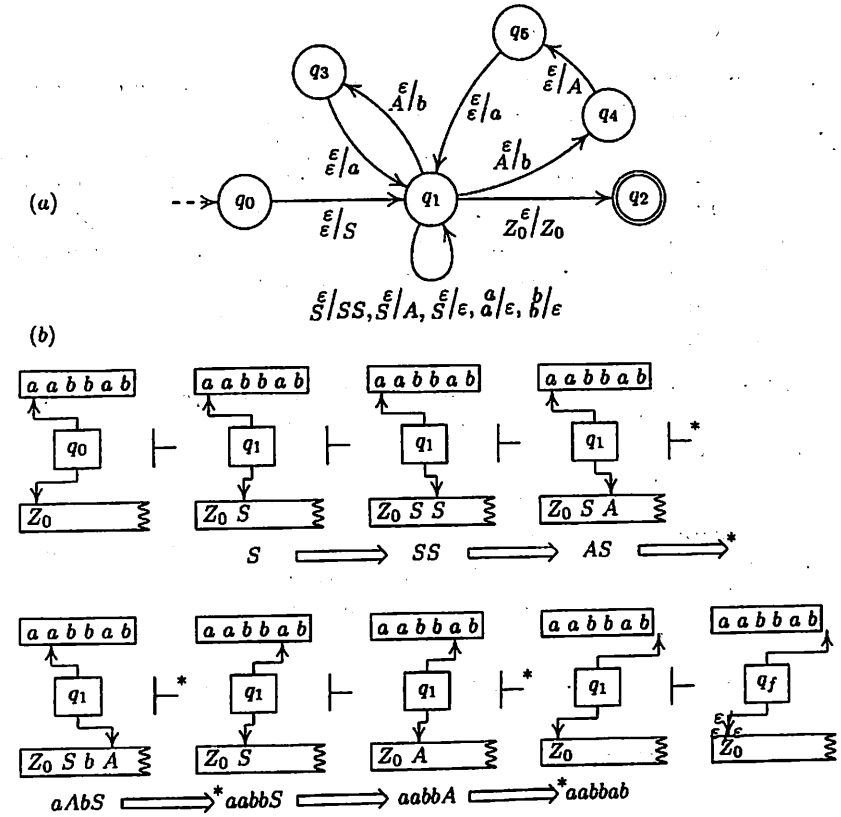


Fig. 3.3.1. (a) A pushdown automaton that accepts the language generated by the grammar of Example 3.3.3.
(b) A leftmost derivation in the grammar and the corresponding computation by the pushdown automaton

aabbab has the leftmost derivation

$$S\Rightarrow SS\Rightarrow AS\Rightarrow aAbS\Rightarrow aabbS\Rightarrow aabbA\Rightarrow aabbab$$

in G. Figure 3.3.1(b) shows the corresponding configurations of M in its computation on such an input. □

From Context-Free Grammars to Recursive Finite-Domain Programs

By Theorems 3.2.1 and 3.3.2 each context-free language is accepted by a recursive finite-domain program. For a given context-free grammar $G=\langle N,\Sigma,P,S\rangle$, the recursive finite-domain program T that accepts $L(G)$ can be of the following form.

T on a given input x nondeterministically traces a leftmost derivation that starts at S. If the leftmost derivation provides the string x, then T accepts its

input. Otherwise, T rejects the input.

T has one procedure for each nonterminal symbol in N, and one procedure for each terminal symbol in Σ. A procedure that corresponds to a nonterminal symbol A is responsible for initiating a tracing of a leftmost subderivation that starts at A. The procedure does so by nondeterministically choosing a production rule of the form $A \rightarrow X_1 ... X_m$, and then calling the procedures that correspond to $X_1,...,X_m$ in the given order. On the other hand, each procedure that corresponds to a terminal symbol is responsible for reading an input symbol and verifying that the symbol is equal to its corresponding terminal symbol.

Each of the procedures above returns the control to the point of invocation, upon successfully completing the given responsibilities. However, each of the procedures terminates the computation at a nonaccepting configuration upon determining that the given responsibility cannot be carried out.

The main program starts a computation by invoking the procedure that corresponds to the start symbol S. Upon the return of control the main program terminates the computation, where the termination is in an accepting configuration if and only if the remainder of the input is empty.

The recursive finite-domain program T can be as depicted in Figure 3.3.2.

```
call S()
if eof then accept
reject

procedure A() /* For each nonterminal symbol A. */
    do
        :
    or
        /* For each production rule of the form A → X1...Xm. */
        call X1()... call Xm()
        return
    or
        :
    until true
end

procedure a() /* For each terminal symbol a. */
    read symbol
    if symbol=a then return
    reject
end
```

Figure 3.3.2. A scheme of recursive finite-domain programs that simulate context-free grammars

Example 3.3.4 If G is the context-free grammar of Example 3.3.1, then $L(G)$ is accepted by the recursive finite-domain program in Figure 3.3.3.

```
call S()
if eof then accept
reject

procedure S()
    do                                          /* S → SS */
        call S() call S() return
    or                                          /* S → A */
        call A() return
    or                                          /* S → ε */
        return
    until true
end

procedure A()
    do                                          /* A → aAb */
        call a() call A() call b() return
    or                                          /* A → ab */
        call a() call b() return
    until true
end

procedure a()
    read symbol
    if symbol=a then return
    reject
end

procedure b()
    read symbol
    if symbol=b then return
    reject
end
```

Figure 3.3.3.
A recursive finite-domain program for the grammar of Example 3.3.1

On input *aabbab* the recursive finite-domain program traces the derivation

$$S \Rightarrow SS \Rightarrow AS \Rightarrow aAbS \Rightarrow aabbS \Rightarrow aabbA \Rightarrow aabbab$$

by calling its procedures in the order indicated in Figure 3.3.4. □

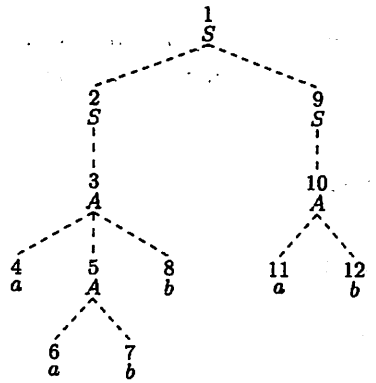


Fig. 3.3.4. The calls to ptocedures that the program of Figire 3.3.3 makes on input *aabbab*

From Recursive Finite-Domain Programs to Context-Free Grammars

A close look at the proof of Theorem 2.3.2 indicates how a given finite-memory program P can be simulated by a Type 3 grammar $G=\langle N,\Sigma,P,S\rangle$.

The grammar uses its nonterminal symbols to record the states of P. Each production rule of the form $A\rightarrow aB$ in the grammar is used to simulate a subcomputation of P that starts at the state recorded by A, ends at the state recorded by B, and reads an input symbol a. However, each production rule of the form $A\rightarrow a$ in the grammar is used to simulate a subcomputation of P that starts at the state that is recorded by A, ends at an accepting state, and reads an input symbol a. The start symbol S of G is used to record the initial state of P. The production rule $S\rightarrow\varepsilon$ is used to simulate an accepting computation of P in which no input value is read.

The proof of the following theorem relies on a similar approach.

Theorem 3.3.3 Every language that is accepted by a recursive finite-domain program is a context-free language.

Proof Consider any recursive finite-domain program P. With no loss of generality it can be assumed that the program has no write instructions. The language that is accepted by P can be generated by a context-free grammar G that simulates the computations of P. The nonterminal symbols of G are used to indicate the start and end states of subcomputations of P that have to be simulated, and the production rules of G are used for simulating transitions between states of P.

The Nonterminal Symbols of G

Specifically, the nonterminal symbols of G consist of

a. A nonterminal symbol A_q, for each state q of P. Each such nonterminal symbol A_q is used for indicating that a subcomputation of P, which starts at state q and ends at an accepting state, has to be simulated. Moreover, each execution of a return instruction in the subcomputation must be for a call that is made previously during the subcomputation.

   The start symbol of G is the nonterminal symbol A_{q_0} that corresponds to the initial state q_0 of P.

b. A nonterminal symbol $A_{q,p}$, for each pair of states q and p corresponding to instruction segments that are in the same procedure of P. Each such nonterminal symbol $A_{q,p}$ is introduced for indicating that a subcomputation, which starts at state q and ends at state p, has to be simulated. In the subcomputation the number of executions of return instructions has to equal the number of executions of call instructions. Moreover, each execution of a return instruction in the subcomputation must be for a call that is made previously during the subcomputation.

The Production Rules of G

The production rules of G consist of

a. A production rule of the form $A_q\rightarrow\alpha A_r$, and a production rule of the form $A_{q,p}\rightarrow\alpha A_{r,p}$, for each q,r,p, and α that satisfy the following condition. The instruction segment that corresponds to state q is neither a call instruction nor a return instruction, and its execution can take the program from state q to state r while reading α.

   A production rule of the form $A_q\rightarrow\alpha A_r$ replaces the objective of reaching an accepting state from state q with the objective of reaching an accepting state from state r.

   A production rule of the form $A_{q,p}\rightarrow\alpha A_{r,p}$ replaces the objective of reaching state p from state q with the objective of reaching state p from state r.

b. A production rule of the form $A_q\rightarrow\varepsilon$, for each state q that corresponds to an **if** *eof* **then accept** instruction.

c. A production rule of the form $A_q\rightarrow A_r$, for each state q that corresponds to a call instruction, where r is the state reached from q. Each such production rule simulates an execution of a **call** which is not matched by an execution of a **return**.

d. A production rule of the form $A_q\rightarrow A_{r,s}A_t$, and a production rule of the form $A_{q,p}\rightarrow A_{r,s}A_{t,p}$, for each q,r,s,t, and p such that the following conditions hold.

   1. State q corresponds to a call instruction whose execution at such a

state causes the program to enter state r.

2. State s corresponds to a return instruction in the called procedure, and the execution of the return instruction at such a state takes the program to state t that is compatible with r.

That is, the subcomputation that starts at state q is decomposed into two subcomputations. One is to be performed by an invoked procedure, starting at state r and ending at state s; the other takes on from the instant that the control returns from the invoked procedure, starting at state t.

e. A production rule of the form $A_{q,q} \rightarrow \varepsilon$ for each state q that corresponds to a return instruction.

Each of the production rules above is used for terminating a successful simulation of a subcomputation performed by an invoked procedure.

$L(G)$ is Contained in $L(P)$

A proof by induction can be used to show that the construction above implies $L(G) = L(P)$.

To show that $L(G)$, is contained in $L(P)$ it is sufficient to show that the following two conditions hold for each string α of terminal symbols.

a. If $A_q \Rightarrow^* \alpha$ in G then P can reach from state q an accepting state while reading α, and in any prefix of the subexecution sequence there must be at least as many executions of call instructions as executions of return instructions.

b. If $A_{q,p} \Rightarrow^* \alpha$ in G, then P can reach state p from state q while reading α. In the subexecution sequence the number of executions of return instructions must equal the number of executions of call instructions, and in any prefix of the subexecution sequence there must be at least as many executions of call instructions as executions of return instructions.

The proof can be by induction on the number of steps i in the derivations. For $i = 1$, the only feasible derivations are those that have either the form $A_q \Rightarrow \varepsilon$ or the form $A_{p,p} \Rightarrow \varepsilon$. In the first case q corresponds to an accept instruction, and in the second case p corresponds to a return instruction. In both cases the subexecution sequences of the program are empty.

For $i > 1$ the derivations must have either of the following forms.

a. $A_q \Rightarrow \alpha_1 A_r \Rightarrow^* \alpha_1 \alpha_2 = \alpha$, or $A_{q,p} \Rightarrow \alpha_1 A_{r,p} \Rightarrow^* \alpha_1 \alpha_2 = \alpha$.

In either case, by definition $A_q \Rightarrow \alpha_1 A_r$ and $A_{q,p} \Rightarrow \alpha_1 A_{r,p}$ correspond to subexecution sequences that start at state q, end at state r, consume the

input α_1, and execute neither a call instruction nor a return instruction. However, by the induction hypothesis $A_r \Rightarrow^* \alpha_2$ and $A_{r,p} \Rightarrow^* \alpha_2$ correspond to subexecution sequences that have the desired properties. Consequently, $A_q \Rightarrow^* \alpha$ and $A_{q,p} \Rightarrow^* \alpha$ also correspond to subexecution sequences that have the desired properties.

b. $A_q \Rightarrow A_{r,s} A_t \Rightarrow^* \alpha_1 \alpha_2$, or $A_{q,p} \Rightarrow A_{r,s} A_{t,p} \Rightarrow^* \alpha_1 \alpha_2$, where $A_{r,s} \Rightarrow^* \alpha_1$.

In either case, by definition q corresponds to a call instruction, r is the state that P reaches from state q, s corresponds to a return instruction, and t is the state that P reaches from state s. However, by the induction hypothesis $A_{r,s} \Rightarrow^* \alpha_1$, $A_t \Rightarrow^* \alpha_2$, and $A_{t,p} \Rightarrow^* \alpha_2$ correspond to subexecution sequences that have the desired properties. Consequently, $A_q \Rightarrow^* \alpha_1 \alpha_2$ and $A_{q,p} \Rightarrow^* \alpha_1 \alpha_2$ also correspond to subexecution sequences that have the desired properties.

$L(P)$ is Contained in $L(G)$

To show that $L(P)$ is contained in $L(G)$ it is sufficient to show that either of the following conditions holds for each subexecution sequence that reads α, starts at state q, ends at state p, and has at least as many executions of return instructions as of call instructions in each of the prefixes.

a. If p corresponds to an accepting state, then G has a derivation of the form $A_q \Rightarrow^* \alpha$.

b. If p corresponds to a return instruction and the subexecution sequence has as many executions of call instructions as of return instructions, then G has a derivation of the form $A_{q,p} \Rightarrow^* \alpha$.

The proof is by induction on the number of moves i in the subexecution sequences. For $i = 0$ the subexecution sequences consume no input, and for them G has the corresponding derivations $A_p \Rightarrow \varepsilon$ and $A_{p,p} \Rightarrow \varepsilon$, respectively.

For $i > 0$ either of the following cases must hold.

a. q does not correspond to a call instruction, or q corresponds to a call instruction that is not matched in the subexecution sequence by a return instruction. In such a case, by executing a single instruction segment the subexecution sequences in question enter some state r from state q while consuming some input α_1.

Consequently, by definition, the grammar G has a production rule of the form $A_q \rightarrow \alpha_1 A_r$ if p is an accepting state, and a production rule of the form $A_{q,p} \rightarrow \alpha_1 A_{r,p}$ if p corresponds to a return instruction.

However, by the induction hypothesis the $i-1$ moves that start in state r have in G a corresponding derivation of the form $A_r \Rightarrow^* \alpha_2$ if p is an

accepting state, and of the form $A_{r,p} \Rightarrow^* \alpha_2$ if p corresponds to a return instruction. α_2 is assumed to satisfy $\alpha_1\alpha_2 = \alpha$.

b. q corresponds to a call instruction that is matched in the subexecution sequence by a return instruction. In such a case the subexecution sequence from state q enters some state r by executing the call instruction that corresponds to state q. Moreover, the subexecution sequence has a corresponding execution of a return instruction that takes the subexecution sequence from some state s to some state t.

Consequently, by definition, the grammar G has a production rule of the form $A_q \rightarrow A_{r,s}A_t$ if p is an accepting state, and a production rule of the form $A_{q,p} \rightarrow A_{r,s}A_{t,p}$ if p corresponds to a return instruction.

However, by the induction hypothesis, the grammar G has a derivation of the form $A_{r,s} \Rightarrow^* \alpha_1$ for the input α_1 that the subexecution sequence consumes between states r and s. In addition, G has either a derivation of the form $A_t \Rightarrow^* \alpha_2$ or a derivation of the form $A_{t,p} \Rightarrow^* \alpha_2$, respectively, for the input α_2 that the subexecution sequence consumes between states t and p, depending on whether p is an accepting state or not. □

Example 3.3.5 Let P be the recursive finite-domain program in Figure 3.3.5(a), with $\{a,b\}$ as a domain of the variables and a as initial value.

(a)

```
call f(x)                /* I1 */
if eof then accept       /* I2 */
reject                   /* I3 */
procedure f(x)
    do                   /* I4 */
        return           /* I5 */
    or
        read x           /* I6 */
        call f(x)        /* I7 */
    until x = a          /* I8 */
end
```


(b)

$$
\begin{array}{rcl}
A_{[1,a]} & \rightarrow & A_{[4,a]} \\
 & \rightarrow & A_{[4,a],[5,a]}A_{[2,a]} \\
 & \rightarrow & A_{[4,a],[5,b]}A_{[2,b]} \\
A_{[2,a]} & \rightarrow & \varepsilon \\
 & \rightarrow & A_{[3,a]} \\
A_{[2,b]} & \rightarrow & \varepsilon \\
 & \rightarrow & A_{[3,b]} \\
A_{[4,a]} & \rightarrow & A_{[5,a]} \\
 & \rightarrow & A_{[6,a]} \\
A_{[4,b]} & \rightarrow & A_{[5,b]} \\
 & \rightarrow & A_{[6,b]} \\
A_{[6,a]} & \rightarrow & aA_{[7,a]} \\
 & \rightarrow & bA_{[7,b]} \\
A_{[6,b]} & \rightarrow & aA_{[7,a]} \\
 & \rightarrow & bA_{[7,b]} \\
A_{[7,a]} & \rightarrow & A_{[4,a],[5,a]}A_{[8,a]} \\
 & \rightarrow & A_{[4,a],[5,b]}A_{[8,b]} \\
A_{[7,b]} & \rightarrow & A_{[4,b],[5,a]}A_{[8,a]} \\
 & \rightarrow & A_{[4,b],[5,b]}A_{[8,b]} \\
A_{[8,b]} & \rightarrow & A_{[4,b]} \\
A_{[4,a],[5,a]} & \rightarrow & A_{[5,a],[5,a]} \\
 & \rightarrow & A_{[6,a],[5,a]} \\
A_{[4,a],[5,b]} & \rightarrow & A_{[5,a],[5,b]} \\
 & \rightarrow & A_{[6,a],[5,b]}
\end{array}
\qquad
\begin{array}{rcl}
A_{[4,b],[5,a]} & \rightarrow & A_{[5,b],[5,a]} \\
 & \rightarrow & A_{[6,b],[5,a]} \\
A_{[4,b],[5,b]} & \rightarrow & A_{[5,b],[5,b]} \\
 & \rightarrow & A_{[6,b],[5,b]} \\
A_{[5,a],[5,a]} & \rightarrow & \varepsilon \\
A_{[5,b],[5,b]} & \rightarrow & \varepsilon \\
A_{[6,a],[5,a]} & \rightarrow & aA_{[7,a],[5,a]} \\
 & \rightarrow & bA_{[7,b],[5,a]} \\
A_{[6,a],[5,b]} & \rightarrow & aA_{[7,a],[5,b]} \\
 & \rightarrow & bA_{[7,b],[5,b]} \\
A_{[6,b],[5,a]} & \rightarrow & aA_{[7,a],[5,a]} \\
 & \rightarrow & bA_{[7,b],[5,a]} \\
A_{[6,b],[5,b]} & \rightarrow & aA_{[7,a],[5,b]} \\
 & \rightarrow & bA_{[7,b],[5,b]} \\
A_{[7,a],[5,a]} & \rightarrow & A_{[4,a],[5,a]}A_{[8,a],[5,a]} \\
 & \rightarrow & A_{[4,a],[5,b]}A_{[8,b],[5,a]} \\
A_{[7,a],[5,b]} & \rightarrow & A_{[4,a],[5,a]}A_{[8,a],[5,b]} \\
 & \rightarrow & A_{[4,a],[5,b]}A_{[8,b],[5,b]} \\
A_{[7,b],[5,a]} & \rightarrow & A_{[4,b],[5,a]}A_{[8,a],[5,a]} \\
 & \rightarrow & A_{[4,b],[5,b]}A_{[8,b],[5,a]} \\
A_{[7,b],[5,b]} & \rightarrow & A_{[4,b],[5,a]}A_{[8,a],[5,b]} \\
 & \rightarrow & A_{[4,b],[5,b]}A_{[8,b],[5,b]} \\
A_{[8,b],[5,a]} & \rightarrow & A_{[4,b],[5,a]} \\
A_{[8,b],[5,b]} & \rightarrow & A_{[4,b],[5,b]}
\end{array}
$$

Figure 3.3.5.
The grammar in (b) generates the language accepted by the program in (a)

$L(P)$ is generated by the grammar G, which has the production rules in Figure 3.3.5(b). $[i,x]$ denotes a state of P that corresponds to the instruction segment I_i, and value x in x.

The derivation tree for the string abb in the grammar G, and the corresponding transitions between the states of the program P on input "a,b,b", are shown in Figure 3.3.6. The symbol $A_{[1,a]}$ states that the computation of P has to start at state $[1,a]$ and end at an accepting state. The production rule $A_{[1,a]} \rightarrow A_{[4,a][5,b]}A_{[2,b]}$ corresponds to a call to f which returns the value b. □

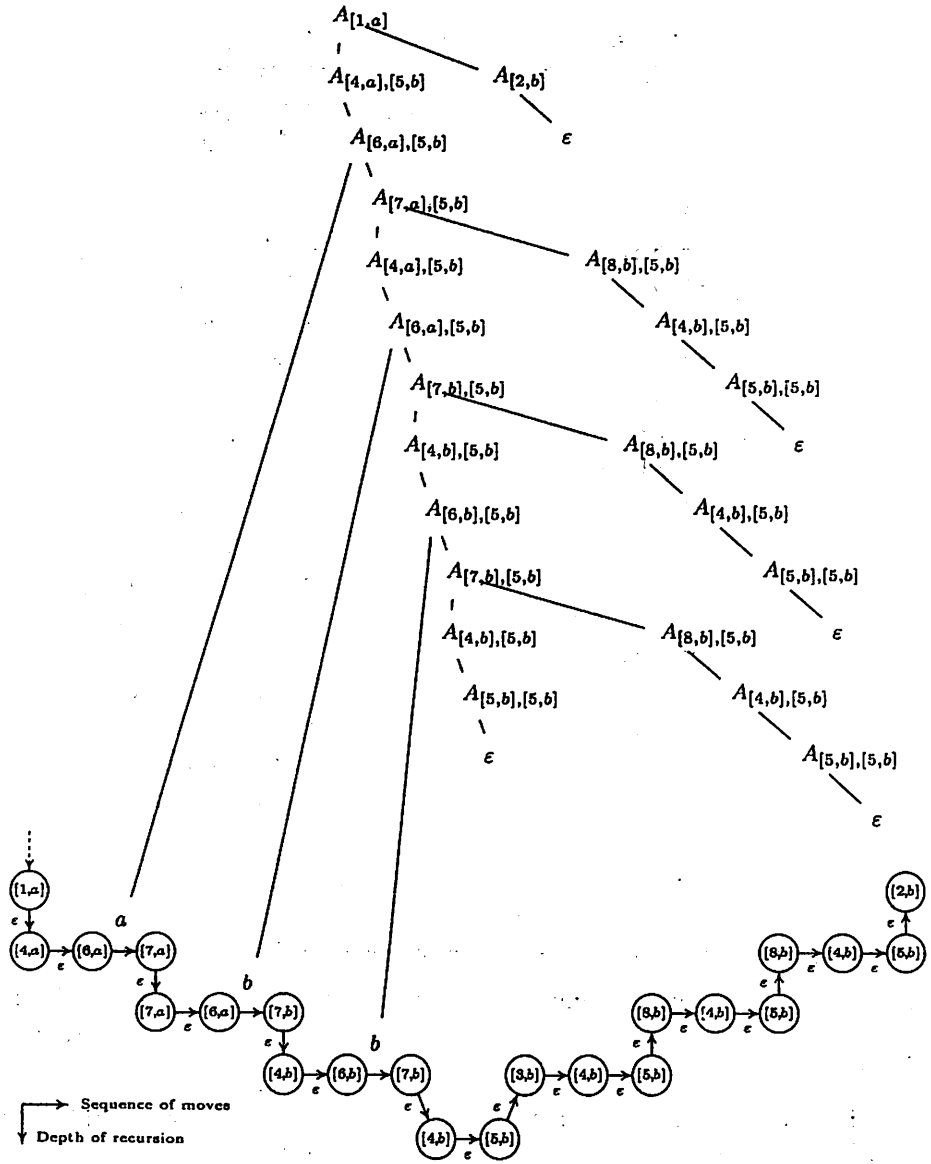


Fig. 3.3.6. A correspondence between a derivation tree and a computation of a recursive finite-domain program

Context-free grammars do not resemble pushdown automata, the way Type 3 grammars resemble finite-state automata. The difference arises because derivations in context-free grammars are recursive in nature, whereas computations of pushdown automata are iterative.

Consequently, some context-free languages can be more easily characterized by context-free grammars, and other context-free languages can be more easily characterized by pushdown automata.

3.4 Limitations of Recursive Finite-Domain Programs

The study of the limitations of finite-memory programs in Section 2.4 relied on the following observation: A subcomputation of an accepting computation of a finite-memory program can be pumped to obtain new accepting computations if the subcomputation starts and ends at the same state. For recursive finite-domain programs similar, but somewhat more complex, conditions are needed to allow pumping of subcomputations.

A Pumping Lemma for Context-Free Languages

The proof of the following theorem uses the abstraction of context-free grammars to provide conditions under which subcomputations of recursive finite-domain programs can be pumped. The corresponding theorem for the degenerated case of finite-memory programs is implied by the choice of $u=v=\varepsilon$.

Theorem 3.4.1 (Pumping lemma for context-free languages) Every context-free language L has a positive integer constant m with the following property. If w is in L and $|w| \geqslant m$, then w can be written as $uvxyz$, where uv^kxy^kz is in L for each $k \geqslant 0$. Moreover, $|vxy| \leqslant m$ and $|vy| > 0$.

Proof Let $G=\langle N,\Sigma,P,S\rangle$ be any context-free grammar. Use t to denote the number of symbols in the longest right-hand side of the production rules of G. With no loss of generality assume that $t \geqslant 2$. Use $|N|$ to denote the number of nonterminal symbols in N. Choose m to equal $t^{|N|+1}$.

Consider any w in $L(G)$ such that $|w| \geqslant m$. Let T denote a derivation tree for w that is minimal for w in the number of nodes. Let π be a longest path from the root to a leaf in T. Let n denote the number of nodes in π.

The number of leaves in T is at most t^{n-1}. Thus, $t^{n-1} \geqslant |w|$ and $|w| \geqslant m = t^{|N|+1}$ imply that $n \geqslant |N|+2$. That is, the path π must have two nodes whose corresponding nonterminal symbols, say E and F, are equal. As a result, w can be written as $uvxyz$, where vxy and x are the strings that correspond to the leaves of the subtrees of T with roots E and F, respectively (see Figure 3.4.1(a)).

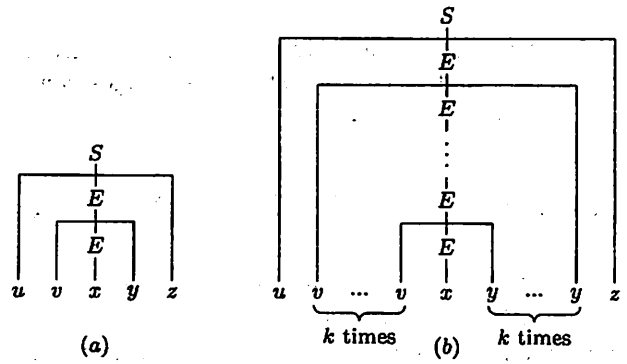


Fig. 3.4.1. (a) A derivation tree T with $E=F$. (b) The derivation tree T_k

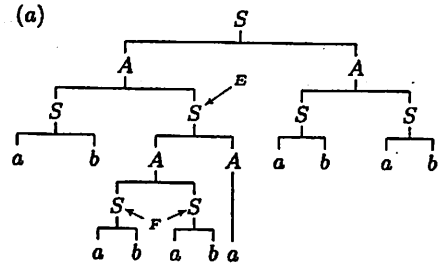
Let T_k be the derivation tree T modified so that the subtree of E, excluding the subtree of F, is pumped k times (see Figure 3.4.1(b)). Then T_k is also a derivation tree in G for each $k \geqslant 0$. It follows that uv^kxy^kz, which corresponds to the leaves of T_k, is also in $L(G)$ for each $k \geqslant 0$.

A choice of E and F from the last $|N|+1$ nonterminal symbols in the path π implies that $|vxy| \leqslant t^{|N|+1} = m$, because each path from E to a leaf contains at most $|N|+2$ nodes. However, $|vy| \neq 0$, because otherwise T_0 would also be a derivation tree for w, contradicting the assumption that T is a minimal derivation tree for w. □

Example 3.4.1 Let $G=\langle N,\Sigma,P,S\rangle$ be the context-free grammar whose production rules are listed below.

$$\begin{aligned} S &\to AA \\ &\to ab \\ A &\to SS \\ &\to a \end{aligned}$$

For G, using the terminology of the proof of the previous theorem, $t=2$, $|N|=2$, and $m=8$. The string $w=(ab)^3a(ab)^2$ has the derivation tree given in Figure 3.4.2(a).



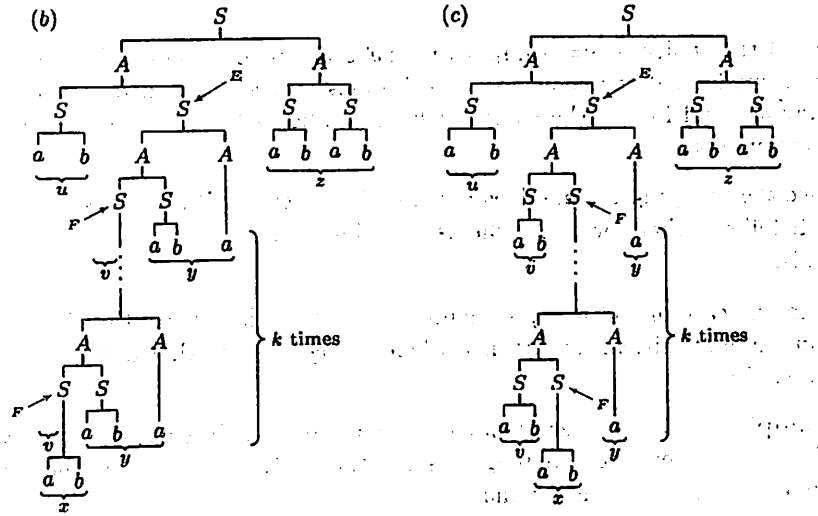


Fig. 3.4.2. (a) A derivation tree T for $(ab)^3a(ab)^2$. (b) A derivation tree T_k for $(ab)^2(aba)^k(ab)^2$. (c) A derivation tree T_k for $ab(ab)^kaba^k(ab)^2$

A longest path in the tree, from the root to a leaf, contains six nodes.

w has two decompositions that satisfy the constraints of the proof of the pumping lemma. One is of the form $u=ab$, $v=\varepsilon$, $x=ab$, $y=aba$, $z=abab$; the other is of the form $u=ab$, $v=ab$, $x=ab$, $y=a$, $z=abab$.

$(ab)^2(aba)^k(ab)^2$ and $ab(ab)^kaba^k(ab)^2$ are the new strings in the language for $k \geqslant 0$, that the proof implies for w by pumping. Figures 3.4.2(b) and 3.4.2(c), respectively, show the derivation trees T_k for these strings. □

Applications of the Pumping Lemma

The pumping lemma for context-free languages can be used to show that a language is not context-free. The method is similar to that for using the pumping lemma for regular languages to show that a language is not regular.

Example 3.4.2 Let L be the language $\{a^nb^nc^n \mid n \geqslant 0\}$. To show that L is not a context-free language, assume to the contrary that L is context-free. Consider the choice of $w=a^mb^mc^m$, where m is the constant implied by the pumping lemma for L.

By the lemma, $a^mb^mc^m$ can be written as $uvxyz$, where $|vxy| \leqslant m$, $|vy| > 0$, and the decomposition satisfies the following conditions.

a. vy contains a's or b's but not c's.
b. vy contains a's or c's but not b's.
c. vy contains b's or c's but not a's.

Moreover, by the pumping lemma, uv^kxy^kz is also in L for each $k \geqslant 0$. However,

for (a) the choice of $k=0$ implies uv^0xy^0z not in L because of too many c's. Similarly, for (b) the choice of $k=0$ implies uv^0xy^0z not in L because of too many b's, and for (c) the choice of $k=0$ implies uv^0xy^0z not in L because of too many a's.

Since the pumping lemma does not hold for $a^mb^mc^m$, it also does not hold for L. It follows, therefore, that the assumption that L is a context-free language is false. □

As in the case of the pumping lemma for regular languages the choice of the string w is of critical importance when trying to show that a language is not context-free.

Example 3.4.3 Consider the language $L=\{\alpha\alpha \mid \alpha \text{ is in } \{a,b\}^*\}$. To show that L is not a context-free language assume the contrary. Let m be the constant implied by the pumping lemma for L.

For the choice $w=a^mb^ma^mb^m$ the pumping lemma implies a decomposition $uvxyz$ such that $|vxy|\leqslant m$ and $|vy|>0$. For such a choice $uv^0xy^0z=uxz=a^ib^ja^sb^t$ with either $i\neq s$ or $j\neq t$. In either case, uxz is not in L. As a result, L cannot be context-free.

On the other hand, for the choice $w=a^mba^mb$ a decomposition $uvxyz$ that satisfies $|vxy|\leqslant m$ and $|vy|>0$ might be of the form $v=y=a^j$ with b in x for some $j>0$. With such a decomposition $uv^kxy^kz=a^{m+(k-1)j}ba^{m+(k-1)j}b$ is also in L for all $k\geqslant 0$. Consequently the latter choice for w does not imply the desired contradiction. □

A Generalization for the Pumping Lemma

The pumping lemma for context-free languages can be generalized to relations that are computable by pushdown transducers. This generalized pumping lemma, in turn, can be used to determine relations that cannot be computed by pushdown transducers.

Theorem 3.4.2 For each relation R that is computable by a pushdown transducer, there exists a constant m such that the following holds for each (w_1,w_2) in R. If $|w_1|+|w_2|\geqslant m$, then w_1 can be written as $u_1v_1x_1y_1z_1$ and w_2 can be written as $u_2v_2x_2y_2z_2$, where

$$(u_1v_1^kx_1y_1^kz_1,u_2v_2^kx_2y_2^kz_2)$$

is also in R for each $k\geqslant 0$. Moreover,

$$|v_1x_1y_1|+|v_2x_2y_2|\leqslant m$$

and

$$|v_1y_1|+|v_2y_2|>0.$$

Proof Consider any pushdown transducer M_1. Let M_2 be the pushdown automaton obtained from M_1 by replacing each transition rule of the form $(q,\alpha,\beta,p,\gamma,\rho)$ with a transition rule of the form $(q,[\alpha,\rho],\beta,p,\gamma)$ if the inequality $[\alpha,\rho]\neq[\varepsilon,\varepsilon]$ hols, and with a transition rule of the form $(q,\varepsilon,\beta,p,\gamma)$ if the equality $[\alpha,\rho]=[\varepsilon,\varepsilon]$ holds. Let h_1 and h_2 be the projection functions defined in the following way:

$$h_1(\varepsilon)=h_2(\varepsilon)=\varepsilon,\quad h_1([\alpha,\rho])=\alpha,\quad h_2([\alpha,\rho])=\rho,$$
$$h_1([\alpha,\rho]w)=h_1([\alpha,\rho])h_1(w),\quad \text{and}$$
$$h_2([\alpha,\rho]w)=h_2([\alpha,\rho])h_2(w).$$

By construction M_2 encodes in its inputs the inputs and outputs of M_1. h_1 and h_2, respectively, determine the values of these encoded inputs and outputs. As a result, (w_1,w_2) is in $R(M_1)$ if and only if w is in $L(M_2)$ for some w such that $h_1(w)=w_1$ and $h_2(w)=w_2$. Use m' to denote the constant implied by the pumping lemma for context-free languages for $L(M_2)$, and choose $m=2m'$.

Consider any (w_1,w_2) in the relation $R(M_1)$ such that $|w_1|+|w_2|\geqslant m$. Then there is some w in the language $L(M_2)$ such that $h_1(w)=w_1$, $h_2(w)=w_2$, and $|w|\geqslant m/2=m'$. By the pumping lemma for context-free languages w can be written as $uvxyz$, where $|vxy|\leqslant m'$, $|vy|>0$, and uv^kxy^kz is in $L(M_2)$ for each $k\geqslant 0$. The result then follows if one chooses

$$u_1=h_1(u),\quad u_2=h_2(u),\quad v_1=h_1(v),\quad v_2=h_2(v),$$
$$x_1=h_1(x),\quad x_2=h_2(x),\quad y_1=h_1(y),\quad y_2=h_2(y),$$
$$z_1=h_1(z),\quad z_2=k_2(z).$$

□

Example 3.4.4 Let M_1 be the pushdown transducer whose transition diagram is given in Figure 3.2.3. Using the terminology of the proof of Theorem 3.4.2, M_2 is the pushdown automaton whose transition diagram is given in Figure 3.4.3.

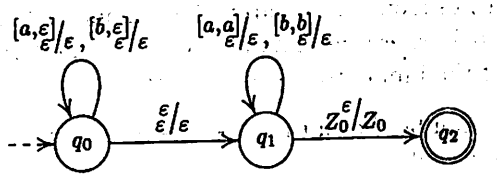


Fig. 3.4.3. A pushdon automaton that "encodes" the pushdown transducer of Figure 3.2.3

The computation of M_1 on input *aabbaa* gives the output *baa*. The computation of M_1 on input *aabbaa* corresponds to the computation of M_2 on input $[a,\varepsilon][a,\varepsilon][b,\varepsilon][b,b][a,a][a,a]$. □

3.5 Closure Properties for Recursive Finite-Domain Programs

By a proof similar to that for Theorem 2.5.1, the class of the relations computable by pushdown transducers, and consequently the class of context-free languages, are closed under union. However, these classes are not closed under intersection and complementation. For instance, the language $\{a^n b^n c^n \mid n \geqslant 0\}$, which is not context-free, is the intersection of the context-free languages $\{a^i b^i c^j \mid i,j \geqslant 0\}$ and $\{a^i b^j c^j \mid i,j \geqslant 0\}$.

Similarly, the class of relations computable by pushdown transducers is not closed under intersection with the relations computable by finite-state transducers. For instance, $\{(a^i b^i c^j, d^i) \mid i,j \geqslant 0\}$ is computable by a pushdown transducer and $\{(a^i b^j c^k, d^k) \mid i,j,k \geqslant 0\}$ is computable by a finite-state transducer. However, the intersection $\{(a^n b^n c^n, d^n) \mid n \geqslant 0\}$ of these two relations cannot be computed by a pushdown transducer.

For context-free languages the following theorem holds.

Theorem 3.5.1 The class of context-free languages is closed under intersection with regular languages.

Proof Consider any pushdown automaton $M_1 = \langle Q_1, \Sigma, \Gamma, \delta_1, q_{01}, Z_0, F_1 \rangle$, and any finite-state automaton $M_2 = \langle Q_2, \Sigma, \delta_2, q_{02}, F_2 \rangle$. With no loss of generality assume that M_2 is ε free and deterministic (see Theorem 2.3.1).

The intersection of $L(M_1)$ and $L(M_2)$ is accepted by the pushdown automaton

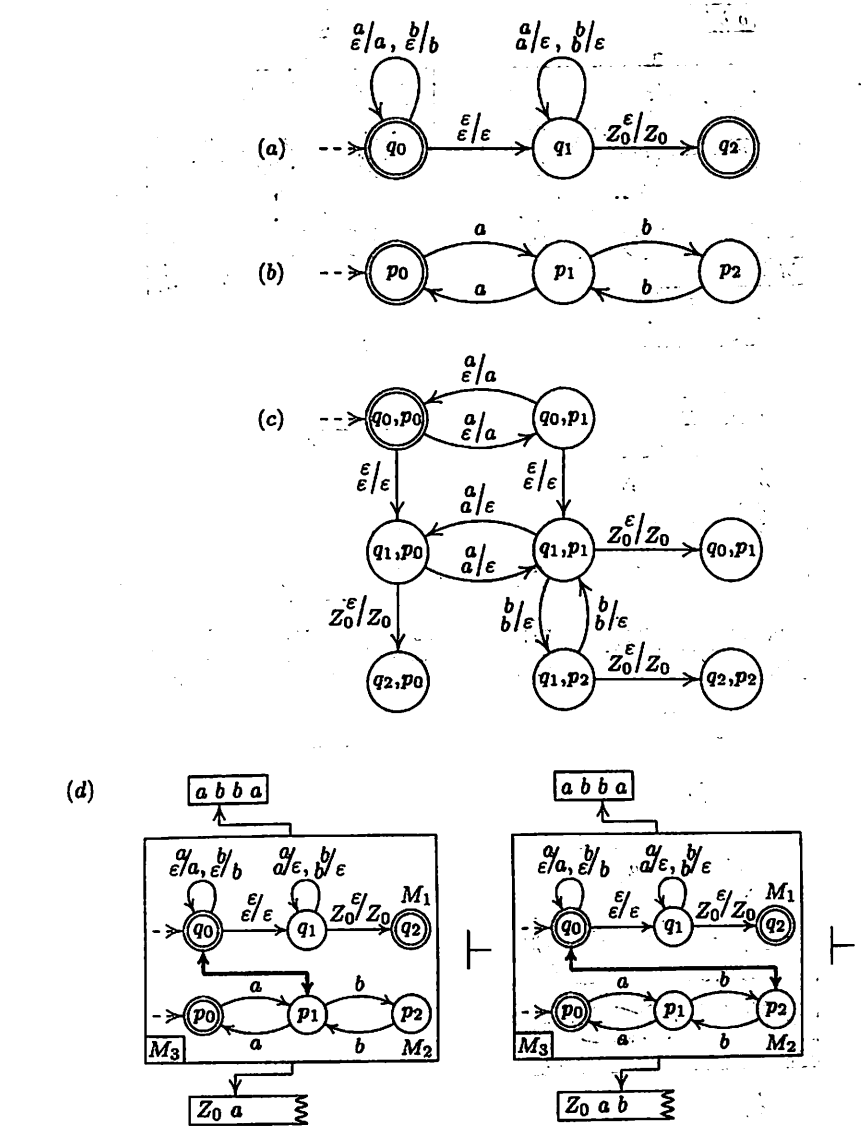
$$M_3 = \langle Q_1 \times Q_2, \Sigma, \Gamma, \delta_3, [q_{01}, q_{02}], Z_0, F_1 \times F_2 \rangle.$$

The transition table δ_3 contains $([q,q'],\alpha,\beta,[p,p'],\gamma)$ if and only if $(q,\alpha,\beta,p,\gamma)$ is in δ_1, and M_2 in zero or one moves can reach state p' from state q' by reading α.

Intuitively, M_3 is a pushdown automaton that simulates the computations of M_1 and M_2 in parallel, where the simulated computations are synchronized to read each symbol of the inputs to M_1 and M_2 together.

By induction on n it can be shown that M_3 accepts an input $a_1 \ldots a_n$ if and only if both M_1 and M_2 accept it. □

Example 3.5.1 The pushdown automaton M_3, whose transition diagram is given in Figure 3.5.1(c),



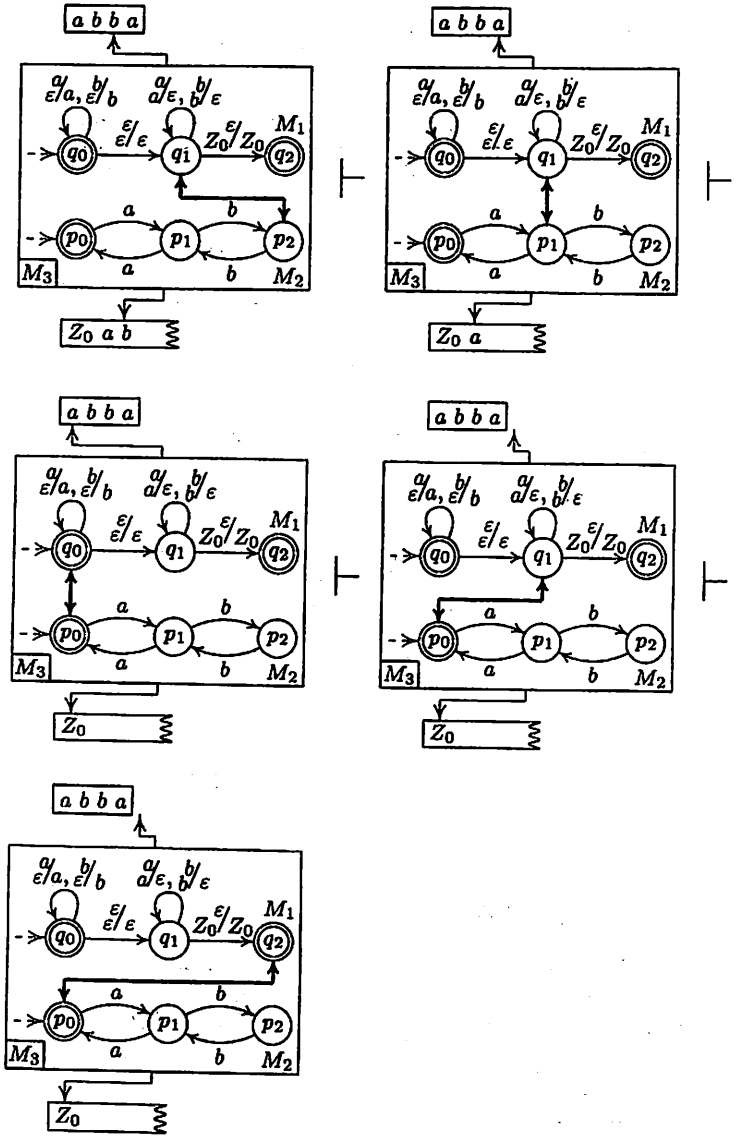


Fig. 3.5.1. The language accepted by the pushdown automaton M_3 in (c) is the intersection of the language accepted by the pushdown automaton M_1 in (a), and the language accepted by the finite-state automaton M_2 in (b). The computation of M_3 on input *abba* is illustrated in (d)

accepts the intersection of the language accepted by the pushdown automaton M_1, whose transition diagram is given in Figure 3.5.1(a), and the language accepted by the finite-state automaton M_2, whose transition diagram is given in Figure 3.5.1(b).

The computation of M_3 on input *abba* is illustrated in Figure 3.5.1(d). □

The languages $\{a^ib^ic^j \mid i,j \geqslant 0\}$ and $\{a^ib^jc^j \mid i,j \geqslant 0\}$ are accepted by deterministic pushdown automata, and the intersection $\{a^nb^nc^n \mid n \geqslant 0\}$ of these languages is not context-free. Consequently, the class of the languages that deterministic pushdown automata accept is not closed under intersection. However, the next theorem will show that the class is closed under complementation. The proof of the theorem uses the following lemma.

Definition A sequence of moves

$$(uq_1v,z_1) \vdash \ldots \vdash (uq_kv,z_k)$$

of a pushdown automaton M is said to be a *loop* if $k>1$, M can move from configuration (uq_1v,z_1) on the same transition rules as from configuration (uq_kv,z_k), and z_1 is a prefix of z_i for $i=2,\ldots,k$. The loop is said to be a *simple loop*, if it contains no loop except itself.

Lemma 3.5.1 Each deterministic pushdown automaton M_1 has an equivalent deterministic pushdown automaton M_2 that halts on all inputs.

Proof Let M_1 be any deterministic pushdown automaton. Let t denote the number of transition rules of M_1. M_1 does not halt on a given input x if and only if it enters a simple loop on x. Moreover, each simple loop of M_1 consists of no more than t^t moves. The desired pushdown automaton M_2 can be constructed from M_1 by employing this observation.

Specifically, M_2 is just M_1 modified to use "marked" symbols in its pushdown store, as well as a counter, say, C in its finite-state control. M_2 marks the topmost symbol in its pushdown store and sets C to zero at the start of each computation, immediately after reading an input symbol, and immediately after removing a marked symbol from the pushdown store. On the other hand, M_2 increases the value of C by one whenever it simulates a move of M_1. Upon reaching a value of t^t+1 in C, the pushdown automaton M_2 determines that M_1 entered a simple loop, and so M_2 halts in a nonaccepting configuration. □

The proof of the following theorem is a refinement of that provided for Theorem 2.5.2, to show that the class of regular languages is closed under complementation.

Theorem 3.5.2 The class of languages that the deterministic pushdown automata accept is closed under complementation.

Proof Consider any deterministic pushdown automaton M. By Lemma 3.5.1 it can be assumed that M has only halting computations, and with no loss of generality it can be assumed that $|\gamma| \leqslant 1$ in each transition rule $(q,\alpha,\beta,p,\gamma)$.

From M to M_{eof}

Let M_{eof} be a deterministic pushdown automaton that accepts $L(M)$, and that in each of its computations halts after consuming all the input. M_{eof} can be constructed from M in the following manner. Let M_{eof} be M initially with an added trap state q_{trap}, and added transition rule of the form $(q_{trap},a,\varepsilon,q_{trap},\varepsilon)$ for each input symbol a. Then repeatedly add to M_{eof} a new transition rule of the form $(q,\alpha,\beta,q_{trap},\beta)$, as long as M_{eof} does not have a next move from state q on input α and topmost pushdown content β.

Elimination of Mixed States

Call a state q of a pushdown automaton a *reading state*, if α is an input symbol in each transition rule $(q,\alpha,\beta,p,\gamma)$ that originates at state q. Call the state q an ε state, if $\alpha=\varepsilon$ in each transition rule $(q,\alpha,\beta,p,\gamma)$ that originates at state q. If the state q is neither a reading state nor an ε state then call it a *mixed state*.

If q is a mixed state of M_{eof}, then each of the transition rules $(q,\alpha,\beta,p,\gamma)$ of M that satisfies $|\alpha|=1$, can be replaced by a pair of transition rules $(q,\varepsilon,\beta,q_\beta,\varepsilon)$ and $(q_\beta,\alpha,\varepsilon,p,\gamma)$, where q_β is a new intermediate, nonaccepting state. Using such transformations M_{eof} can be modified to include no mixed states.

A Modification to M_{eof}

M_{eof} can be further modified to obtain a similar deterministic pushdown automaton M_{eof_max}, with the only difference being that upon halting, M_{eof_max} is in a reading state. The modification can be done in the following way.

a. Let M_{eof_max} initially be M_{eof}.

b. Rename each state q of M to $[q,accept]$ if it is an accepting state, and to $[q,reject]$ if it is a nonaccepting state.

c. As long as the pushdown automaton M_{eof_max} has a transition rule of the form $([q,accept],\varepsilon,\beta,[p,reject],\gamma)$, replace it with a transition rule of the form $([q,accept],\varepsilon,\beta,[p,accept],\gamma)$. In addition, if $[p,accept]$ is a new state, then for each transition rule of the form $([p,reject],\alpha,\beta',p',\gamma')$ add also

   1. A transition rule of the form $([p,accept],\alpha,\beta',p',\gamma')$ if $p'\neq[p,reject]$ or $\alpha\neq\varepsilon$.

   2. A transition rule of the form $([p,accept],\alpha,\beta',[p,accept],\gamma')$, if $p'=[p,reject]$ and $\alpha=\varepsilon$.

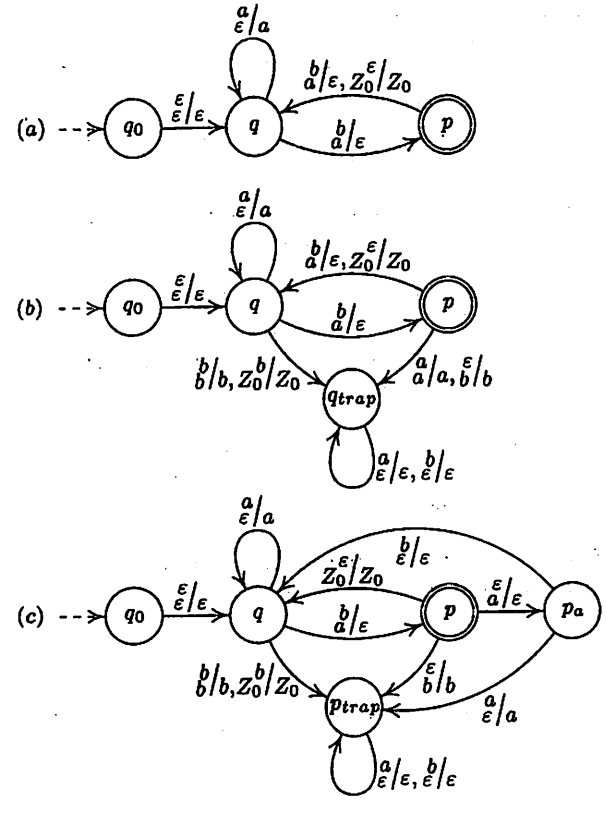
d. Let a state of M_{eof_max} be an accepting state if and only if it is a reading state of the form $[q,accept]$.

The above transformations propagate the "accepting property" of ε states until their "blocking" reading states.

A Pushdown Automaton That Accepts the Complementation of $L(M)$

The constructed pushdown automaton M_{eof_max} on a given input has a unique sequence of moves that ends at a reading state after consuming all the input. The sequence of moves remains the same, even when a different subset of the set of reading states is chosen to be the set of accepting states. Thus, the deterministic pushdown automaton that accepts the complementation of $L(M)$ can be obtained from M_{eof_max}, by requiring that the reading states of the form $[q,reject]$ become the accepting states. □

Example 3.5.2 Let M be the deterministic pushdown automaton whose transition diagram is given in Figure 3.5.2(a).



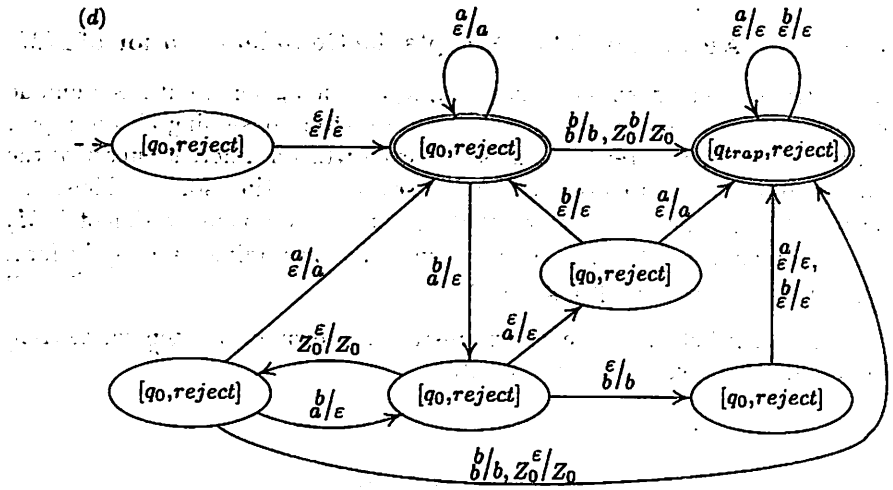


Fig. 3.5.2 (a). A pushdown automaton M. (b) The complementation M_{eof} for M (c) M_{eof} modified to include no mixed states (d) A pushdown automaton that accepts the complementation of $L(M)$

Using the terminology in the proof of Theorem 3.5.2, the state q_0 of M is an ϵ state, the state q is a reading state, and the state p is a mixed state.

The transition diagram of M_{eof} is given in Figure 3.5.2(b). The transition diagram of M_{eof} modified to include no mixed states, is given in Figure 3.5.2(c). The transition diagram in Figure 3.5.2(d) is of a deterministic pushdown automaton that accepts the complementation of $L(M)$. □

The closure under complementation of the class of the languages that deterministic pushdown automata accept, the nonclosure of the class under intersection, and DeMorgan's law all imply the nonclosure of the class under union.

Corollary 3.5.1 There are languages that are accepted by nondeterministic pushdown automata, but that cannot be accepted by any deterministic pushdown automata.

3.6 Decidable Properties for Recursive Finite-Domain Programs

The first theorem of this section provides a generalization of the decidability of the emptiness problem for finite-state automata.

Theorem 3.6.1 The emptiness problem is decidable for pushdown automata.

Proof Consider any pushdown automaton $M_1 = \langle Q, \Sigma, \Gamma, \delta_1, q_0, Z_0, F \rangle$. Let c be a new symbol not in Σ. Let δ_2 be δ_1 with each transition rule of the form

$(q, \varepsilon, \beta, p, \gamma)$ being replaced with a transition rule of the form (q, c, β, p, γ). Let M_2 be the pushdown automaton $\langle Q, \Sigma \cup \{c\}, \Gamma, \delta_2, q_0, Z_0, F \rangle$.

Intuitively, we see that M_2 is the pushdown automaton M_1 modified to read the symbol c whenever M_1 is to make a move that reads no input symbol. By construction, M_1 can reach configuration (uqv, w) in t moves if and only if there exists u_c such that M_2 can reach configuration $(u_c qv, w)$ in t moves, where u_c is a string obtained from u by insertion of some c's and $|u_c| = t$. Thus, $T(M_1) = \varnothing$ if and only if $T(M_2) = \varnothing$.

Denote m as the constant that the pumping lemma for context-free languages implies for $L(M_2)$. The shortest string x in $L(M_2)$ cannot be longer than m. Otherwise, a contradiction would arise because by the pumping lemma if x is in $L(M_2)$ and if its length is at least m, then a shorter string is also in $L(M_2)$.

On input x the pushdown automaton M_2 can have at most $|x|$ moves. Consequently, the emptiness of $L(M_2)$ or, equivalently, of $L(M_1)$ can be checked by considering all the possible execution sequences of M_2 or, equivalently, of M_1 that consist of no more than m moves. □

The decidability of the emptiness problem for pushdown automata can be used for showing the decidability of some problems for finite-state transducers. One such example is the decidability of the equivalence problem for deterministic finite-state transducers. For the general class of finite-state transducers as well as the class of pushdown automata the problem is undecidable (Corollary 4.7.1 and Corollary 4.7.2, respectively). On the other hand, for deterministic pushdown automata and for deterministic pushdown transducers the problem is open.

Corollary 3.6.1 The equivalence problem is decidable for deterministic finite-state transducers.

Proof Consider any two deterministic finite-state transducers M_1 and M_2. From M_1 and M_2 a finite-state automaton M_3 can be constructed such that M_3 accepts the empty set if and only if $L(M_1) = L(M_2)$. The construction can be as in the proof of Theorem 2.6.4.

On the other hand, one can also construct from M_1 and M_2 a pushdown automaton M_4 that accepts a given input if and only if both M_1 and M_2 accept it, while providing different outputs. That is, M_4 accepts the empty set if and only if M_1 and M_2 agree in their outputs on the inputs that they both accept.

A computation of M_4 on a given input consists of simulating in parallel, as in the proof of Theorem 3.5.1, the computations of M_1 and M_2 on such an input. The simulation is in accordance with either of the following cases, where the choice is made nondeterministically.

Case 1 M_4 simulates accepting computations of M_1 and M_2 that provide outputs of different lengths. During the simulation, M_4 ignores the outputs of M_1 and M_2. However, at each instant of the simulation, the pushdown store of M_4 holds the absolute value of the difference between the length of the outputs produced so far by M_1 and M_2. M_4 accepts the input if and only if it reaches accepting states of M_1 and M_2 at the end of the input, with a nonempty pushdown store.

Case 2 M_4 simulates accepting computations of M_1 and M_2 that provide outputs differing in their jth symbol, for some j that is no greater than their lengths. The simulation is similar to that in Case 1. The main difference is that M_4 records in the pushdown store the changes in the length of the output of M_i only until it establishes (nondeterministically) that M_i reached its jth output symbol, $i=1,2$. In addition, M_4 records in its finite-state control the jth output symbols of M_1 and M_2. Upon completing the simulation, M_4 accepts the input if and only if its pushdown is empty and the recorded symbols in the finite-state control are distinct.

Given M_3 and M_4, a pushdown automaton M_5 can then be constructed to accept $L(M_3) \cup L(M_4)$. M_5 accepts the empty set if and only if M_1 and M_2 are equivalent. The result thus follows from Theorem 3.6.1. □

The uniform halting problem is undecidable for pushdown automata (Corollary 4.7.3). However, the decidability of the emptiness problem for pushdown automata can be used to show the decidability of the uniform halting problem for deterministic pushdown automata.

Theorem 3.6.2 The uniform halting problem is decidable for deterministic pushdown automata.

Proof Consider any deterministic pushdown automaton M_1. From M_1 a deterministic pushdown automaton M_2, similar to that in the proof of Lemma 3.5.1, can be constructed. The only difference is that here M_2 accepts a given input if and only if it determines that M_1 reaches a simple loop. By construction, M_2 accepts an empty set if and only if M_1 halts on all inputs. □

The proof of the last theorem fails for nondeterministic pushdown automata because accepting computations of nondeterministic pushdown automata can include simple loops, without being forced to enter an infinite loop.

Theorem 3.6.3 The halting problem is decidable for pushdown automata.

Proof Consider any pair (M,x) of a pushdown automaton M and of an input x for M. From x, a finite-state automaton M_x can be constructed that accepts only the input x. However, from M, a pushdown automaton M_1 can be constructed to accept a given input if and only if M has a sequence of transition

rules that leads M to a simple loop on the input. The construction can be similar to the proof of Theorem 3.6.2.

From M and M_x, a pushdown automaton $M_{a,x}$ can be constructed that accepts the intersection of $L(M)$ with $L(M_x)$ (see Theorem 3.5.1). By construction, $M_{a,x}$ accepts a nonempty set if and only if M accepts x. By Theorem 3.6.1 it can be determined if $M_{a,x}$ accepts a nonempty set. If so, then M is determined to halt on input x. Otherwise, in a similar way, a pushdown automaton $M_{1,x}$ can be constructed to accept the intersection of $L(M_1)$ and $L(M_x)$. By construction, $M_{1,x}$ accepts the empty set if and only if M has only halting computations on input x. The result then follows from Theorem 3.6.1. □

3.7 Exercises

3.1.1 For each of the following relations give a recursive finite-domain program that computes the relation.

a. $\{(a^ib^j,c^i) \mid i \geqslant 0\}$

b. $\{(xy,x) \mid xy \text{ is in } \{a,b\}^* \text{ and } |x|=|y|\}$

c. $\{(x,y) \mid x \text{ and } y \text{ are in } \{0,1\}^*,\ |x|=|y|, \text{ and } y \neq x^{rev}\}$

3.2.1 For each of the following relations give a (deterministic, if possible) pushdown transducer that computes the relation.

a. $\{(a^ib^j,a^jb^i) \mid i,j \geqslant 0\}$

b. $\{(x,a^ib^j) \mid x \text{ is in } \{a,b\}^*,\ i=(\text{number of } a\text{'s in } x), \text{ and } j=(\text{number of } b\text{'s in } x)\}$

c. $\{(xyz,xy^{rev}z) \mid xyz \text{ is in } \{a,b\}^*\}$

d. $\{(a^ib^j,c^k) \mid i \leqslant k \leqslant j\}$

e. $\{(a^ib^j,c^k) \mid k=min(i,j)\}$

f. $\{(w,c^k) \mid w \text{ is in } \{a,b\}^*, \text{ and } k=min(\text{number of } a\text{'s in } w, \text{ number of } b\text{'s in } w)\}$

g. $\{(xy,yx^{rev}) \mid x \text{ and } y \text{ are in } \{a,b\}^*\}$

h. $\{(x,x^{rev}x) \mid x \text{ is in } \{a,b\}^*\}$

i. $\{(x,y) \mid x \text{ and } y \text{ are in } \{a,b\}^*, \text{ and } y \text{ is a permutation of } x\}$

3.2.2 Find a pushdown transducer that simulates the computations of the recursive finite-domain program of Figure 3.E.1. Assume that the variables have the domain $\{0,1\}$, and the initial value 0.

```
call RP(x)                    /* I1 */
if eof then accept            /* I2 */
reject                        /* I3 */
procedure RP(y)
    read x                    /* I4 */
    if x≠y then               /* I5 */
        call RP(x)            /* I6 */
    write y                   /* I7 */
    return                    /* I8 */
end
```

Figure 3.E.1

3.2.3 For each of the following languages find a (deterministic, if possible) pushdown automaton that accepts the language.

a. $\{vww^{rev} \mid v \text{ and } w \text{ are in } \{a,b\}^*, \text{ and } |w|>0\}$

b. $\{x \mid x \text{ is in } \{a,b\}^* \text{ and each prefix of } x \text{ has at least as many } a\text{'s as } b\text{'s}\}$

c. $\{a^ib^ja^jb^i \mid i,j>0\}$

d. $\{w \mid w \text{ is in } \{a,b\}^*, \text{ and } w \neq w^{rev}\}$

e. $\{xx^{rev} \mid x \text{ is accepted by the finite-state automaton of Figure 3.E.2}\}$

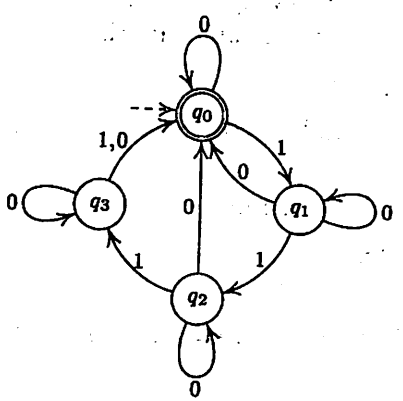


Fig. 3.E.2.

f. $\{x \mid x = x^{rev} \text{ and } x \text{ is accepted by the finite-state automaton of Figure 3.E.2}$

3.3.1 For each of the following languages construct a context-free grammar that generates the language.

a. $\{x\#y \mid x \text{ and } y \text{ are in } \{a,b\}^* \text{ and have the same number of } a\text{'s}\}$

b. $\{a^ib^jc^k \mid i \neq j \text{ or } j \neq k\}$

c. $\{x \mid x \text{ is in } \{a,b\}^* \text{ and each prefix of } x \text{ has at least as many } a\text{'s as } b\text{'s}\}$

d. $\{x\#y \mid x \text{ and } y \text{ are in } \{a,b\}^* \text{ and } y \text{ is not a permutation of } x\}$

e. $\{x\#y \mid x \text{ and } y \text{ are in } \{a,b\}^* \text{ and } x \neq y\}$

3.3.2 Find a Type 2 grammar that is equivalent to the context-free grammar $G=\langle N,\Sigma,P,S\rangle$, whose production rules are given in Figure 3.E.3(a).

$S \to CD$			
$\to a$	$S \to AB$		
$C \to \varepsilon$	$A \to BAB$	$S \to aASa$	
$\to SC$	$\to a$	$\to b$	$S \to aA$
$\to b$	$B \to ABA$	$A \to aAa$	$A \to Sb$
$D \to CC$	$\to b$	$\to b$	$\to ab$
(a)	(b)	(c)	(d)

Figure 3.E.3

3.3.3 Let $G=\langle N,\Sigma,P,S\rangle$ be the context-free grammar whose production rules are listed in Figure 3.E.3(b). Find a recursive finite-domain program and a pushdown automaton that accept the language generated by G.

3.3.4 Let M be the pushdown automaton whose transition diagram is given in Figure 3.E.4.

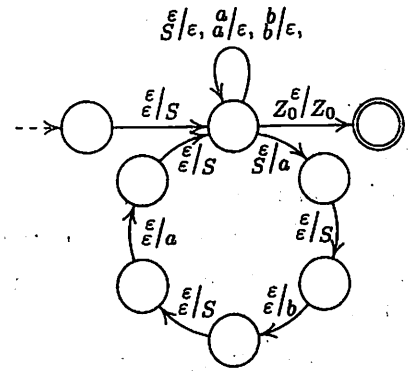


Fig. 3.E.4.

Find a context-free grammar that generates $L(M)$.

3.3.5 Find a deterministic pushdown automaton that accepts the language generated by the grammar $G=\langle N,\Sigma,P,S\rangle$, whose production rules are given in Figure 3.E.3(c).

3.3.6 Let the program P and the grammar G be as in Example 3.3.5. Find a derivation in G that corresponds to an accepting computation of P on input *bab*.

3.3.7 Find the context-free grammar that accepts the same language as the program P in Figure 3.E.5, according to the proof of Theorem 3.3.3. Assume that the domain of the variables is equal to $\{a,b\}$, with a as initial value.

```
do                              /* I1 */
    call f(x)                   /* I2 */
    if eof then accept          /* I3 */
until false                     /* I4 */

procedure f(x)
    if x=b then                 /* I5 */
        return                  /* I6 */
    read x                      /* I7 */
    call f(x)                   /* I8 */
    return                      /* I9 */
end
```

Figure 3.E.5

3.4.1 Redo Example 3.4.1 for the case that G has the production rules listed in Figure 3.E.3(d) and $w=a^5b^4$.

3.4.2 Show that each of the following sets is not a context-free language.

a. $\{a^nb^lc^t \mid t>l>n>0\}$

b. $\{\alpha\alpha^{rev}\alpha \mid \alpha \text{ is in } \{a,b\}^*\}$

c. $\{\alpha\beta\alpha^{rev}\beta^{rev} \mid \alpha \text{ and } \beta \text{ are in } \{a,b\}^*\}$

d. $\{a^n\alpha a^n\alpha \mid \alpha \text{ is in } \{a,b\}^*, \text{ and } n=(\text{the number of } a\text{'s in } \alpha)\}$

e. $\{\alpha\#\beta \mid \alpha \text{ and } \beta \text{ are in } \{a,b\}^* \text{ and } \beta \text{ is a permutation of } \alpha\}$

f. $\{\alpha\beta \mid$ the finite-state transducer whose transition diagram is given in Figure 3.E.6 has output β on input $\alpha\}$

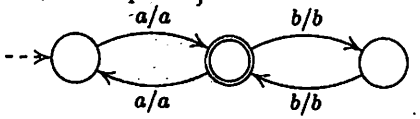


Fig. 3.E.6.

g. $\{a^{n!} \mid n \geq 1\}$

3.4.3 Show that the relation $\{(x,d^n) \mid x \text{ is in } \{a,b,c\}^* \text{ and } n=min(\text{number of } a\text{'s in } x, \text{ number of } b\text{'s in } x, \text{ number of } c\text{'s in } x)\}$ is not computable by a pushdown transducer.

3.5.1 Show that the class of the relations computable by pushdown transducers is closed under each of the following operations Ψ:

a. Inverse, that is, $\Psi(R)=R^{-1}=\{(y,x) \mid (x,y) \text{ is in } R\}$.

b. Composition, that is, $\Psi(R_1,R_2)=\{(x,y) \mid x=x_1x_2 \text{ and } y=y_1y_2 \text{ for some } (x_1,y_1) \text{ in } R_1 \text{ and some } (x_2,y_2) \text{ in } R_2\}$.

c. Reversal, that is, $\Psi=\{(x^{rev},y^{rev}) \mid (x,y) \text{ is in } R\}$.

3.5.2 Show that the class of context-free languages is not closed under the operation $\Psi(L_1,L_2)=\{xyzw \mid xz \text{ is in } L_1 \text{ and } yw \text{ is in } L_2\}$.

3.5.3 Find a pushdown automaton that accepts the intersection of the language accepted by the pushdown automaton whose transition diagram is given in Figure 3.E.7(a), and the language accepted by the finite-state automaton whose transition diagram is given in Figure 3.5.1(b).

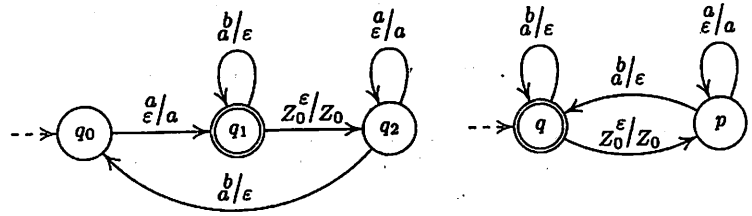


Fig. 3.E.7.

3.5.4 Let M be the deterministic pushdown automaton given in Figure 3.E.7(b). Find the pushdown automaton that accepts the complementation of $L(M)$ in accordance with the proof of Theorem 3.5.2.

3.5.5 Show that if a relation is computable by a deterministic pushdown transducer, then its complementation is computable by a pushdown transducer.

3.6.1 Show that the membership problem is decidable for pushdown automata.

3.6.2 Show that the single valuedness problem is decidable for finite-state transducers.

3.6.3 Show that the equivalence problem for finite-state transducers is reducible to the equivalence problem for pushdown automata.

3.8 Bibliographic Notes

McCarthy (1963) introduced recursion to programs. Recursive finite-domain programs and their relationship to pushdown transducers were considered in Jones and Muchnick (1978). The pushdown automata were introduced by Oettinger (1961) and Schutzenberger (1963). Evey (1963) introduced the pushdown transducers. The equivalence of pushdown automata to context-free languages were observed by Chomsky (1962) and Evey (1963). The pumping lemma for context-free languages is from Bar-Hillel, Perles, and Shamir (1961). Scheinberg (1960) used similar arguments to show that $\{a^n b^n c^n \mid n \geqslant 1\}$ is not context-free. The closure of context-free languages under union, and their non-closure under intersection and complementation, were noticed by Scheinberg (1960). The closure of the class of context-free languages under composition and under intersection with regular languages is due to Bar-Hillel, Perles, and Shamir (1961). Schutzenberger (1963) showed the closure under complementation of the class of languages that are accepted by the deterministic pushdown automata (Theorem 3.5.2). Bar-Hillel, Perles, and Shamir (1961) showed the closure of context-free languages under reversal (see Exercise 3.5.1(c)).

The decidability of the emptiness problem for context-free grammars is also due to Bar-Hillel, Perles, and Shamir (1961). The decidability of the equivalence problem for the deterministic finite-state transducers in Corollary 3.6.1 follows from Bird (1973). The proof technique used here is from Gurari (1979). This proof technique coupled with proof techniques of Valiant (1973) were used by Ibarra and Rosier (1981) to show the decidability of the equivalence problem for some subclasses of deterministic pushdown transducers.

Greibach (1981) and Hopcroft and Ullman (1979) provide additional insight into the subject.

BIBLIOGRAPHY

1. Adleman, L. (1978). "Two theorems on random polynomial time," *Proceedings of the 19th IEEE Symposium on Foundations of Computer Science*, 75-83.
2. Aho,A., Hopcroft, J., and Ullman, J. (1974). *The Design and Analysis of Computer Algorithms*, Reading, MA: Addison-Wesley.
3. Aho,A., Sethi, R., and Ullman, J. (1986). *Compilers: Principles, Techniques, and Tools*, Reading, MA: Addison-Wesley.
4. Allender, E. (1986). "Characterizations of PUNC and precomputation," *International Colloquium on Automata, Languages and Programming*, Lecture Notes in Computer Science 226, Berlin: Springer-Verlag, 1-10.
5. Bar-Hillel, Y., Perles, M., and Shamir, E. (1961). "On formal properties of simple phrase structure grammars", *Zeitschrift fr Phonetik Sprachwissenschaft und Kommunikations-forschung 14*, 143-172.
6. Batcher, K. (1968). "Sorting networks and their applications," *Proceedings of the 32nd AFIPS Spring Joint Computer Conference*, 307-314.
7. Bird, M. (1973). "The equivalence problem for deterministic two-tape automata," *Journal of Computer and Systems Sciences* 7, 218-236.
8. Borodin, A. (1977). "On relating time and space to size and depth," *SIAM Journal on Computing 6*, 733-744.
9. Bchi,J. (1960). "Weak second-order arithmetic and finite automata," *Zeitschrift fur math. Logik und Grundlagen d. Math. 6*, 66-92.
10. Chandra, A., Stockmeyer, L., and Vishkin, U. (1984). "Constant depth reducibilities," *SIAM Journal on Computing 13*, 423-439.
11. Chomsky, N. (1959). "On certain formal properties of grammars," *Information and Control 2*, 137-167.
12. Chomsky, N. (1962). "Context-free grammars and pushdown storage," *Quarterly Progress Report 65*, MIT Research Laboratories of Electronics, 187-194.
13. Chomsky, N., and Miller, G. (1958). "Finite-state languages," *Information and Control* 1,91-112.
14. Chomsky, N., and Schutzenberger, M. (1963). "The algebraic theory of context free languages," *Computer Programming and Formal Systems*, 118-161.
15. Church, A. (1936). "An unsolvable problem of elementary number theory," *American Journal of Mathematics 58*, 345-363.
16. Cobham, A. (1964). "The intrinsic computational difficulty of functions," *Proceedings of the 1964 Congress for Logic, Mathematics and the Philosophy of Science*, Amsterdam: North Holland, 24-30.
17. Cook, S. (1971). "The complexity of theorem-proving procedures," *Proceedings of the 3rd Annual ACM Symposium on Theory of Computing*, 151-158.
18. Cook, S. (1981). "Towards a complexity theory of synchronous parallel computations," *L'Enseignement Mathematique 27*, 99-124.
19. Cook, S. (1983). "The classification of problems which have fast parallel algorithms," *Proceedings of the 4th International Foundations of Computer Science Conference*, Lecture Notes in Computer Science 158, Berlin: Springer Verlag, 78-93.

20. Cook, S., and Reckhov, R. (1973). "Time bounded random access machines," *Journal of Computer and Systems Sciences 7*, 354-375.
21. Danthine, A. (1980). "Protocol representation with finite state models," *IEEE Transactions on Communications 4*, 632-643.
22. DeLeeuw, K., Moore, E., Shannon, C., and Shapiro, N. (1956). "Computability by probabilistic machines," *Automata Studies*, Princeton, NJ: Princeton University Press, 183-212.
23. Edmonds, J. (1965a. "Path, trees and flowers," *Canadian Journal of Mathematics 17*, 449-467.
24. Edmonds, J. (1965b. "Minimum partition of matroid in independent subsets," *Journal of Research of the National Bureau of Standard Sect69B*, 67-72.
25. Ehrenfeucht, A., Parikh, R., and Rozenberg, G. (1981). "Pumping lemmas for regular sets," *SIAM Journal on Computing 10*, 536-541.
26. Evey, J. (1963). "Application of pushdown store machines," *Proceedings 1963 Fall Joint Computer Conference*, Montvale, NJ:AFIPS Press, 215-227.
27. Floyd, R. (1967). "Nondeterministic algorithms," *Journal of the Association for Computing Machinery 14*, 636-644.
28. Fortune, S., and Wyllie, J. (1978). "Parallelism in random access machines," *Proceedings of the 10th Annual ACM Symposium on Theory of Computing*, 114-118.
29. Freivalds, R. (1979). "Fast probabilistic algorithms," *Proceedings of the 1979 Mathematical Foundations of Computer Science*, Lecture Notes in Computer Science 74, Berlin: Springer-Verlag, 57-69.
30. Garey, M., and Johnson, D. (1979). *Computers and Intractability: A Guide to the Theory of NP-Completeness*, San Francisco, CA:W.H. Freeman and Company.
31. Gill, J. (1977). "Computational complexity of probabilistic Turing machines," *SIAM Journal on Computing 6*, 675-694.
32. Goldschlager, L. (1982). "A unified approach to models of synchronous parallel machines," *Journal of the Association for Computing Machinery 29*, 1073-1086.
33. Greibach, S. (1981). "Formal languages: Origins and directions," *Annals of the History of Computing 3*, 14-41.
34. Griffiths, T. (1968). "The unsolvability of the equivalence problem for Λ-free nondeterministic generalized machines," *Journal of the Association for Computing Machinery 15*, 409-413.
35. Grolmusz, V., and Ragde, P. (1987). "Incomparability in parallel computation," *Proceedings of the 28th IEEE Symposium on Foundations of Computer Science*, 89-98.
36. Gurari, E. (1979). "Transducers with decidable equivalence problem," Technical Report, University of Wisconsin-Milwaukee, 1979. Revised version, State University of New York at Buffalo, 1981.
37. Harrison, M. (1978). *Introduction to Formal Language Theory*, Reading, MA: Addison-Wesley.

38. Hartmanis, J., and Stearns, R. (1965). "On the computational complexity of algorithms," *Transactions of the American Mathematical Society 117*, 285-306.
39. Hilbert, D. (1901). "Mathematical problems," *Bulletin of the American Mathematical Society 8*, 437-479.
40. Hopcroft, J., and Ullman, J. (1969). "Some results on tape bounded Turing machines," *Journal of the Association for Computing Machinery 16*, 168-177.
41. Hopcroft, J., and Ullman, J. (1979). *Introduction to Automata Theory, Languages and Computation*, Reading, MA: Addison-Wesley.
42. Hunt, H. (1973). "On the time and type complexity of languages," *Proceedings of the 5th Annual ACM Symposium on Theory of Computing*, 10-19.
43. Hunt, H., Constable, R., and Sahni, S. (1980). "On the computational complexity of scheme equivalence," *SIAM Journal on Computing 9*, 396-416.
44. Ibarra, O., and Rosier, L. (1981). "On the decidability of equivalence for deterministic pushdown transducers," *Information Processing Letters 13*, 89-93.
45. Immerman, N. (1987). "Space is closed under complementation," Technical Report, New Haven, CT: Yale University.
46. Hong, J. (1985). "On similarity and duality of computation," *Information and Control 62*, 109-128.
47. Johnson, D. (1983). "The NP-completeness column: An ongoing guide," *Journal of Algorithms 4*, 189-203.
48. Johnson, D. (1984). "The NP-completeness column: An ongoing guide," *Journal of Algorithms 5*, 433-447.
49. Jones, N., and Laaser, W., (1976). "Complete problems for deterministic polynomial time," *Theoretical Computer Science 3*, 105-118.
50. Jones, N., and Muchnick, S. (1977). "Even simple programs are hard to analyze," *Journal of the Association for Computing Machinery 24*, 338-350.
51. Jones, N., and Muchnick, S. (1978). "The complexity of finite memory programs with recursion," *Journal of the Association for Computing Machinery 25*, 312-321.
52. Kannan, R. (1982). "Circuit-size lower bounds and non-reducibility to sparse sets," *Information and Control 55*, 40-56.
53. Karp, R. (1972). "Reducibility among combinatorial problems," *Complexity of Computer Computations*, edited by R. Mille and J. Thatcher, New York: Plenum Press, 85-104.
54. Kindervater, G., and Lenstra, J. (1985). "Parallel Algorithms," in *Combinatorial Optimization: Annotated Bibliographies*, edited by M.O'hEigeartaigh,J. Lenstra, and A. Rinnooy Kan, New York: John Wiley and Sons, 106-128.
55. Kleene, S. (1956). "Representation of events in nerve nets and finite automata," *Automata Studies*, Princeton, NJ: Princeton University Press, 3-41.
56. Kučera,K. (1982). "Parallel computation and conflicts in memory access," *Information Processing Letters 14*, 93-96. A correction, *ibid 17*, 107.
57. Kuroda, S. (1964). "Classes of languages and linear bounded automata," *Information and Control 7*, 207-223.

58. Ladner, R. (1975). "The circuit value problem is log space complete for P," *Sigact News 7*, 18-20.

59. Landweber, P. (1963). "Three theorems on phrase structure grammars of Type 1," *Information and Control 6*, 131-136.

60. Lesk, M. (1975). "LEX - a lexical analyzer generator," Technical Report 39, Murray Hill, NJ: Bell Laboratories.

61. Levin, L. (1973). "Universal sorting problems," *Problemi Peredaci Informacii 9*, 115-116. English translation in *Problems of Information Transmission 9*, 265-266.

62. Lucker, G. (1975). "Two NP-complete problems in non-negative integer programming," Report No. 178, Computer Science Laboratory, Princeton, NJ: Princeton University.

63. Maffioli, F., Speranza, M., and Vercellis, C. (1985). "Randomized algorithms," in *Combinatorial Optimization - Annotated Bibliographies*, edited by M.O'hEigeartaigh, J. Lenstra, and A. Rinnooy Man, New York: John Wiley and Sons, 89-105.

64. Matijasevic, Y. (1970). "Enumerable sets are Diophantine," *Doklady Akademiky Nauk SSSR191*, 279-282. English translation: *Soviet Math Doklady 11*, 354-357.

65. McCarthy, J. (1963). "A basis for a mathematical theory of computation," *Computer Programming and Formal Systems*, edited by P. Braffort and D., Hirschberg, Amsterdam: North-Holland.

66. McCulloch, W., and Pitts, W. (1943). "A logical calculus of the ideas immanent in nervous activity," *Bulletin of Mathematical Biophysics 5*, 115-133.

67. Megiddo, N. (1981). "Applying parallel computation algorithms in the design of serial algorithms," *Proceedings of the 22nd IEEE Symposium on Foundations of Computer Science*, 399-408.

68. Meyer, A., and Ritchie, R. (1967). "The complexity of loop programs," *Proceedings of the ACM National Meeting*, 465-469.

69. Moore, E. (1956). "Gedanken experiments on sequential machines," *Automata Studies*, Princeton, NJ: Princeton University Press, 129-153.

70. Muller, D., and Preparata, F. (1975). "Bounds to complexities of networks for sorting and for switching," *Journal of the Association for Computing Machinery 22*, 195-201.

71. Myhill, J. (1957). "Finite automata and the representation of events," *WADD TR-57-624*, Dayton, OH: Wright Patterson Air Force Base.

72. Myhill, J. (1960). "Linear bounded automata," *WADD TR-60-165*, Dayton, OH: Wright Patterson Air Force Base.

73. Oettinger, A. (1961). "Automatic syntactic analysis and the pushdown store," *Proceedings of the 12th Symposia in Applied Mathematics*, Providence, RI: American Mathematical Society, 104-109.

74. Pippenger, E. (1979). "On simultaneous resource bounds," *Proceedings of the 20th IEEE Symposium on Foundations of Computer Science*, 307-311.

75. Post, E. (1946). "A variant of a recursively unsolvable problem," *Bulletin of the American Mathematical Society 52*, 264-268.

76. Rabin, M. (1976). "Probabilistic algorithms," *Algorithms and Complexity - New Directions and Recent Results*, edited by J. Traub, New York: Academic- Press, 21-29.

77. Rabin, M., and Scott, D. (1959). "Finite automata and their decision problems," *IBM Journal of Research and Development 3*, 114-125.

78. Ritchie, R. (1963). "Classes of predictably computable functions," *Transactions of the American Mathematical Society 106*, 139-173.

79. Ruzzo, L. (1981). "On uniform circuit complexity," *Journal of Computer and Systems Sciences 22*, 365-383.

80. Savitch, W. (1970). "Relationships between nondeterministic and deterministic tape complexities," *Journal of Computer and Systems Sciences 4*, 177-192.

81. Scheinberg, S. (1960). "Note on the Boolean properties of context free languages," *Information and Control 3*, 372-375.

82. Schutzenberger, M. (1963). "On context-free languages and pushdown automata," *Information and Control 6*, 246-264.

83. Schwartz, J. (1980). "Fast probabilistic algorithms for verification of polynomial identities," *Journal of the Association for Computing Machinery 27*, 701-717.

84. Shannon, C. (1949). "The synthesis of two terminal switching circuts," *Bell System Technical Journal 28*, 59-98.

85. Sheperdson, J. (1959). "The reduction of two-way automata to one-way automata," *IBM Journal of Research and Development 3*, 198-200.

86. Shiloach, Y., and Vishkin, U. (1981). "Finding the maximum, merging and sorting in a parallel computation model," *Journal of Algorithms 2*, 88-102.

87. Sipser, M. (1978). "Halting bounded computations," *Proceedings of the 19th IEEE Symposium on Foundations of Computer Science*, 73-74.

88. Solovay, R. and Strassen, V. (1977). "A fast Monte Carlo test for primality," *SIAM Journal on Computing 6*, 84-85. A correction, *ibid 7*, 118.

89. Stearns, R., Hartmanis, J., y y nand Lewis, P. (1965). "Hierarchies of memory limited computations," *Proceedings of the 6th Annual IEEE Symposium on Switching Circuit Theory and Logical Design*, 191-202.

90. Stockmeyer, L. (1985). "Classifying the computational complexity of problems," *IBM Research Report*, San Jose, CA.

91. Szelepcsenyi, R. (1987). "The method of forcing for nondeterministic automata," *Bulletin of the European Association for Theoretical Computer Science 33*, 96-100.

92. Turing, A. (1936). "On computable numbers with an application to the Entscheidungs problem," *Proceedings of the London Mathematical Society 2*, 230-265. A correction, *ibid*, 544-546.

93. Valiant, L. (1973). "Decision procedures for families of deterministic pushdown automata," Ph.D. Thesis, University of Warwick, U.K.

94. Valiant, L. (1975). "Parallelism in comparison problems," *SIAM Journal on Computing 4*, 348-355.

95. Welsh, D. (1983). "Randomised algorithms," *Discrete applied Mathematics 5*, 133-145.

Table of Contents

PREFACE 3

1 GENERAL CONCEPTS 5

1.1 Alphabets, Strings, and Representations 6
1.2 Formal Languages and Grammars 10
1.3 Programs 21
1.4 Problems 37
1.5 Reducibility among Problems 44
1.6 Exercises 46
1.7 Bibliographic Notes 54

2 FINITE-MEMORY PROGRAMS 55

2.1 Motivation 55
2.2 Finite-State Transducers 58
2.3 Finite-State Automata and Regular Languages 74
2.4 Limitations of Finite-Memory Programs 85
2.5 Closure Properties for Finite-Memory Programs 90
2.6 Decidable Properties for Finite-Memory Programs 94
2.7 Exercises 96
2.8 Bibliographic Notes 101

3 RECURSIVE FINITE-DOMAIN PROGRAMS 103

3.1 Recursion 103
3.2 Pushdown Transducers 107
3.3 Context-Free Languages 126
3.4 Limitations of Recursive Finite-Domain Programs 141
3.5 Closure Properties for Recursive Finite-Domain Programs 146
3.6 Decidable Properties for Recursive Finite-Domain Programs 152
3.7 Exercises 155
3.8 Bibliographic Notes 160

BIBLIOGRAPHY 161

Fundamental Structures of Computer Science
(Course Materials)

Part 1
Introduction to Automata and Formal Languages

Тираж 100 экз. Объем 168 стр.
Формат 60x84/16. Бумага офсетная №1.
Плотность 80 г/м^2. Печать RISO.

ТОО «Эверо». Полиграфические услуги.
Тел.: 39-32-69, тел./факс: 32-38-43
e-mail: evero@nursat.kz