Graduate Texts in Mathematics

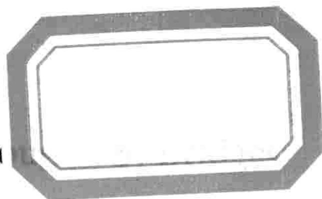
Computability

A Mathematical Sketchbook

可计算性

Douglas S . Bridges

Springer-Verlag
世界图书出版公司



D(

Computability

A Mathematical Sketchbook

With 29 Illustrations

Springer-Verlag

世界图书出版公司

北京·广州·上海·西安

书　　名: Computability
作　　者: D. S. Bridges
中 译 名: 可计算性
出 版 者: 世界图书出版公司北京公司
印 刷 者: 北京世图印刷厂
发　　行: 世界图书出版公司北京公司(北京朝内大街137号　100010)
开　　本: 大32　　印张: 6
版　　次: 1997年9月第1版　　1997年9月第1次印刷
书　　号: ISBN 7-5062-3309-6/O·188
版权登记: 图字 01-97-0457
定　　价: 42.00元

世界图书出版公司北京公司已获得Springer-Verlag授权在中国境内独家重印、发行。

此为试读,需要完整PDF请访问: www.ertongbook.com

For Vivien, Iain, Hamish, and Catriona

'I can't believe that!' said Alice. 'Can't you?' the Queen said in a pitying tone. 'Try again: draw a long breath and shut your eyes.' Alice laughed. 'There's no use trying,' she said: 'One can't believe impossible things.' 'I daresay you haven't had much practice,' said the Queen.

LEWIS CARROLL, *Through the Looking Glass.*

Preface

My intention in writing this book is to provide mathematicians and mathematically literate computer scientists with a brief but rigorous introduction to a number of topics in the abstract theory of computation, otherwise known as *computability theory* or *recursion theory*. It develops major themes in computability, such as Rice's Theorem and the Recursion Theorem, and provides a systematic account of Blum's abstract complexity theory up to his famous Speed-up Theorem.

A relatively unusual aspect of the book is the material on computable real numbers and functions, in Chapter 4. Parts of this material are found in a number of books, but I know of no other at the senior/beginning graduate level that introduces elementary recursive analysis as a natural development of computability theory for functions from natural numbers to natural numbers.[1] This part of the book is definitely for mathematicians rather than computer scientists and has a prerequisite of a first course in elementary real analysis; it can be omitted, without rendering the subsequent chapters unintelligible, in a course including the more standard topics in computability theory found in Chapters 4-6.

I believe, against the trend towards weighty, all-embracing treatises (*vide* the typical modern calculus text), that many mathematicians would like to be able to purchase books that give them insight into unfamiliar branches of the subject in a relatively short compass and without requiring a major investment of time, effort, or money. Following that belief, I have had to exclude from this book many topics—such as detailed proofs of the equivalence of various mathematical models of computation, the theory of degrees of unsolvability, and polynomial and nonpolynomial complexity—whose absence will be deplored by at least some of the experts in the field. I hope that my readers will be inspired to pursue their study of recursion theory in such major works as [9, 24, 28, 29].

A number of excellent texts on computability theory are primarily aimed at computer scientists rather than mathematicians, and so do not always maintain the level of rigour that would be expected in a modern text on, say, abstract algebra. I have tried to maintain that higher level of rigour

[1] Some of the work in this book—notably, Proposition (4.28) and the application of the Recursion Theorem preceding Exercises (5.14)—appears to be original.

throughout, even at the risk of deflecting the interest of mathematically insecure computer scientists.

Ideally, all mathematics and computer science majors should be exposed to at least some of the material found in this book. It horrifies me that in some universities such majors can still graduate ignorant of the theoretical limitations of the computer, as expressed, for example, by the undecidability of the halting problem (Theorem (4.2)). A short course on computability, accessible even to students below junior level, would comprise Chapters 1-3 and the material in Chapter 4 up to Exercises (4.7). A longer course for more advanced undergraduates would also include Rice's theorem and the Recursion Theorem, from Chapter 5, and at least parts of Chapter 6. The entire book, including the difficult material on recursive analysis from Chapter 4, would be suitable for a course for bright seniors or beginning graduate students.

I have tried to make the book suitable for self-study. To this end, it includes solutions for most of the exercises. Those exercises for which no solutions are given have been marked with the asterisk (*); of varying levels of difficulty, they provide the instructor with material for homework and tests. *The exercises form an integral part of the book* and are not just there for the student's practice; many of them develop material that is used in later proofs, which is another reason for my inclusion of solutions.

My interest in constructive mathematics [5] leads me to comment here on the logic of computability theory. This is *classical logic,* the logic used by almost all mathematicians in their daily work. However, the use of classical logic has some perhaps undesirable consequences. Consider the following definition of a function f on the set $\mathbf{N}$ of natural numbers: for all n, $f(n)$ equals 1 if the Continuum Hypothesis is true, and equals 0 if the Continuum Hypothesis is false.[2] Since 'most mathematicians are formalists on weekdays and Platonists on Sundays', at least on Sundays most of us would accept this as a good definition of a function f. According to classical logic, f is computable because there exists an algorithm that computes it; that algorithm is either the one which, applied to any natural number n, outputs 1, or else the one which, applied to any natural number n, outputs 0. But the Continuum Hypothesis is independent of the axioms of ZFC (Zermelo-Fraenkel set theory plus the axiom of choice), the standard framework of mathematics, so we will never be able to tell, using ZFC alone, which of the two algorithms actually is the one that computes f.

It appears from this example, eccentric though it may be, that the standard theory of computation does not exactly match computational practice,

[2] The **Continuum Hypothesis** (CH) says that the smallest cardinal number greater than $\aleph_0$, the cardinality of $\mathbf{N}$, is $2^{\aleph_0}$, the cardinality of the set of all subsets of $\mathbf{N}$. The work of Cohen [13] and Gödel [17] shows that neither CH nor its negation can be proved within Zermelo-Fraenkel set theory plus the axiom of choice; see also [3], pages 420-428.

in which we would expect to pin down the algorithms that we use. A facetious question may reinforce my point: what would happen to an employee who, in response to a request that he write software to perform a certain computation, presented his boss with two programs and the information that, although one of those programs performed the required computation, nobody could ever tell which one?

With classical logic there seems to be no way to distinguish between functions that are computed by programs which we can pin down and those that are computable but for which there is no hope of our telling which of a range of programs actually performs the desired computation. To handle this problem successfully, we need a different logic, one capable of distinguishing between *existence in principle* and *existence in practice.* For example, with constructive (intuitionistic) logic the problem disappears,[3] since f is then not properly defined: it is only properly defined if we can decide the truth or falsehood of the Continuum Hypothesis (which we cannot) and therefore which of the two possible algorithms computes f.

Having said this, let me stress that, despite the inability of classical logic to make certain distinctions of the type I have just dealt with, *I have followed standard practice and used classical logic throughout this book.*

Not only the logic but also most of the material that I have chosen is standard, although some of the exercises and examples are new. I have drawn on a number of books, including [34] for the treatment of Turing machines in Chapter 1; [20] for the first parts of Chapters 4 and 5; and [9, 14, 29] for parts of Chapter 7.

The origins of my book lie in courses I gave at the University of Buckingham (England), New Mexico State University (USA), and the University of Waikato (New Zealand). I am grateful to the students in those classes for the patience with which they received various slowly improving draft versions.[4] Special thanks are due to Fred Richman for many illuminating conversations about recursion theory; to Paul Halmos for his advice and encouragement; and to Cris Calude, Nick Dudley Ward, Graham French, Hazel Locke, and Steve Merrin, all of whom have read versions of the text and made many helpful corrections and suggestions. As always, it is my wife and children who suffered most as the prolonged birth of this work took so much of my care and attention; I present the book to them with love and gratitude.

May 1993 *Douglas S. Bridges*

[3] For a development of computability theory using intuitionistic logic see Chapter 3 of [8].

[4] The first drafts of this book were prepared using the T^3 *Scientific Word Processing System.* The final version was produced by converting the drafts to $\TeX$ and then using *Scientific Word.* T^3 and *Scientific Word* are both products of TCI Software Research, Inc. The diagrams were drawn with *Aldus Freehand* v. 3.1 (©Aldus Corporation).

Douglas S. Bridges
Department of Mathematics
University of Waikato
Private Bag 3105
Hamilton, New Zealand

Editorial Board

J.H. Ewing
Department of Mathematics
Indiana University
Bloomington, IN 47405
USA

F.W. Gehring
Department of Mathematics
University of Michigan
Ann Arbor, MI 48109
USA

P.R. Halmos
Department of Mathematics
Santa Clara University
Santa Clara, CA 95053
USA

Mathematics Subject Classifications (1991): 03Dxx

Library of Congress Cataloging-in-Publication Data
Bridges, D.S. (Douglas S.), 1945–
Computability : a mathematical sketchbook / Douglas S. Bridges.
p. cm. — (Graduate texts in mathematics)
Includes bibliographical references and index.
ISBN 0-387-94174-6
1. Computable functions. I. Title. II. Series.
QA9.59.B75 1994
511.3—dc20 93-21313

Printed on acid-free paper.

© 1994 Springer-Verlag New York, Inc.
All rights reserved. This work may not be translated or copied in whole or in part without the written permission of the publisher (Springer-Verlag New York, Inc., 175 Fifth Avenue, New York, NY 10010, USA), except for brief excerpts in connection with reviews or scholarly analysis. Use in connection with any form of information storage and retrieval, electronic adaptation, computer software, or by similar or dissimilar methodology now known or hereafter developed is forbidden.
The use of general descriptive names, trade names, trademarks, etc., in this publication, even if the former are not especially identified, is not to be taken as a sign that such names, as understood by the Trade Marks and Merchandise Marks Act, may accordingly be used freely by anyone.

This reprint has been authorized by Springer-Verlag (Berlin/Heidelberg/New York) for sale in the People's Republic of China only and not for export therefrom.
Reprinted in China by Beijing World Publishing Corporation, 1997.

ISBN 0-387-94174-6 Springer-Verlag New York Berlin Heidelberg
ISBN 3-540-94174-6 Springer-Verlag Berlin Heidelberg New York

Contents

	Preface	**vii**
	Preliminaries	**1**
1	**What Is a Turing Machine?**	**5**
2	**Computable Partial Functions**	**19**
3	**Effective Enumerations**	**35**
4	**Computable Numbers and Functions**	**47**
5	**Rice's Theorem and the Recursion Theorem**	**75**
6	**Abstract Complexity Theory**	**93**
	Solutions to Exercises	**117**
	Solutions for Chapter 1	118
	Solutions for Chapter 2	120
	Solutions for Chapter 3	130
	Solutions for Chapter 4	136
	Solutions for Chapter 5	156
	Solutions for Chapter 6	166
	References	**173**
	Index	**176**

此为试读,需要完整PDF请访问: www.ertongbook.com

Preliminaries

Throughout this book we assume familiarity with the standard notations and basic results of informal set theory, as found in [18]. We use the following notation for sets of numbers.

The set of natural numbers:	$\mathbf{N} \equiv \{0, 1, 2, \ldots\}$.
The set of rational numbers:	$\mathbf{Q} \equiv \{\pm m/n : m, n \in \mathbf{N},\ n \neq 0\}$.
The set of real numbers:	$\mathbf{R}$.

For $n \geq 1$ we write X^n for the n–fold Cartesian product $X \times X \times \cdots \times X$ (n factors) of X, and P_i^n for the ith projection of X^n—that is, the mapping from X^n onto X defined by[1]

$$P_i^n(x_1, \ldots, x_n) \equiv x_i.$$

We denote by $(x_n)_{n=0}^{\infty}$, or $(x_0, x_1, \ldots)$, or even just (x_n), the sequence whose terms are indexed by $\mathbf{N}$ and whose n^{th} term is x_n.

We shall be particularly interested in what happens when a computer is programmed to compute natural number outputs from inputs in $\mathbf{N}^n$. Since the execution of a program may fail to terminate when the machine is run with certain inputs—for example, a program for computing the reciprocal of a natural number will not normally output a natural number if it is run with the input 0—we are forced to deal with functions that are defined on subsets of $\mathbf{N}^n$ and not necessarily on the entire set $\mathbf{N}^n$. This leads us to the notion of a **partial function** φ from a set A to a set B : that is, a function φ whose domain is a subset of A and which takes values in B; the domain of φ may be empty and is usually not the entire set A. We refer to such a function φ as the partial function $\varphi : A \to B$; we write domain(φ) for its domain, and range(φ) for its range. We also say that $\varphi(x)$ is **defined** if $x \in$ domain(φ), and that $\varphi(x)$ is **undefined** if $x \in A$ and $x \notin$ domain(φ). A partial function from A to B whose domain is the entire set A is called, oxymoronically, a **total partial function** from A to B.

There is an unwritten convention (not followed by all authors) that uses Greek letters to denote partial functions and Roman letters to denote total ones. We shall usually follow that convention, although some partial func-

[1] The symbol $\equiv$ means *is defined as* or *is identical to.*

Preliminaries

Throughout this book we assume familiarity with the standard notations and basic results of informal set theory, as found in [18]. We use the following notation for sets of numbers.

The set of natural numbers: $\mathbf{N} \equiv \{0, 1, 2, \ldots\}$.
The set of rational numbers: $\mathbf{Q} \equiv \{\pm m/n : m, n \in \mathbf{N},\ n \neq 0\}$.
The set of real numbers: $\mathbf{R}$.

For $n \geq 1$ we write X^n for the n-fold Cartesian product $X \times X \times \cdots \times X$ (n factors) of X, and P_i^n for the ith projection of X^n—that is, the mapping from X^n onto X defined by[1]

$$P_i^n(x_1, \ldots, x_n) \equiv x_i.$$

We denote by $(x_n)_{n=0}^{\infty}$, or $(x_0, x_1, \ldots)$, or even just (x_n), the sequence whose terms are indexed by $\mathbf{N}$ and whose n^{th} term is x_n.

We shall be particularly interested in what happens when a computer is programmed to compute natural number outputs from inputs in $\mathbf{N}^n$. Since the execution of a program may fail to terminate when the machine is run with certain inputs—for example, a program for computing the reciprocal of a natural number will not normally output a natural number if it is run with the input 0—we are forced to deal with functions that are defined on subsets of $\mathbf{N}^n$ and not necessarily on the entire set $\mathbf{N}^n$. This leads us to the notion of a **partial function** φ from a set A to a set B : that is, a function φ whose domain is a subset of A and which takes values in B; the domain of φ may be empty and is usually not the entire set A. We refer to such a function φ as the partial function $\varphi : A \to B$; we write domain(φ) for its domain, and range(φ) for its range. We also say that $\varphi(x)$ is **defined** if $x \in$ domain(φ), and that $\varphi(x)$ is **undefined** if $x \in A$ and $x \notin$ domain(φ). A partial function from A to B whose domain is the entire set A is called, oxymoronically, a **total partial function** from A to B.

There is an unwritten convention (not followed by all authors) that uses Greek letters to denote partial functions and Roman letters to denote total ones. We shall usually follow that convention, although some partial func-

---

[1] The symbol $\equiv$ means *is defined as* or *is identical to.*

where $\varphi : \mathbf{N} \to \mathbf{N}$ is a partial function, we imply that $\varphi(n)$ *is defined* and less than or equal to k.

For computational purposes a natural number n is usually represented by a string of symbols drawn from some suitable set. For example, 5 may be represented by the string *aaaaa* whose symbols are drawn from the singleton set $\{a\}$, by the binary string 101, by the single decimal digit 5, and so on. Strings appear so frequently in the early chapters of our book that it is a good idea to give a formal definition of them here.

By a **string of length** n over the set X we mean an element $(x_1, \ldots, x_n)$ of the n-fold Cartesian product $X^n \equiv X \times \cdots \times X$ (n factors); the elements $x_1, \ldots, x_n$ are called the **terms** of the string, x_k being the kth term. When we consider $(x_1, \ldots, x_n)$ as a string over X, we usually omit the parentheses and commas, and simply write $x_1 \ldots x_n$. We assume that there is a unique **empty string** Λ of length 0 over X; informally, Λ is the unique string with no terms over X. We denote by X^* the set of strings over X, and by $|u|$ the length of the string $u \in X^*$.

Strings u, v over X can be combined to form a string $u \cdot v$, usually written uv, by the operation of **concatenation.** Informally, this involves writing one string next to another. The following is a formal inductive definition: for all strings u, v over X, and all elements x of X,

$$\begin{aligned} \Lambda \cdot u &\equiv u, \\ (xu) \cdot v &\equiv x(u \cdot v). \end{aligned}$$

It is a simple exercise in induction to show that concatenation has the properties you would expect it to have. For example, the length of uv is the sum of the lengths of u and v; $u(vw) = (uv)w$ (so we write either side as uvw); and

$$\Lambda u = u = u\Lambda.$$

In the context of computability and formal language theory a nonempty finite set X is often called an **alphabet**, and a subset of X^* a **language** over X. (The set of words defined in the *Oxford English Dictionary* is a language over the alphabet $\{a, b, c, \ldots, z\}$; British readers might argue that this is *the* English language!) The following are useful constructions with languages A, B over a finite alphabet X :

- The **concatenation** of A and B :

$$A \cdot B \equiv \{uv : u \in A,\ v \in B\}.$$

- The **iterate** (or **Kleene star**) of A :

$$A^* \equiv \{u_1 u_2 \ldots u_n : n \geq 0,\ \forall k\,(u_k \in A)\}.$$

In this context the union of A and B is often written $A + B$, rather than $A \cup B$.

For example, if $X = \{0,1,2\}$, $A = \{0,1\}^*$, and $B = \{2\}$, then $A \cdot B$ consists of the string 2, together with all strings of the form $x_1 \dots x_n 2$ with $x_i \in \{0,1\}$ for $1 \le i \le n$; $A^* = A$, and B^* consists of all finite (possibly empty) strings with each term equal to 2; and $A + B$ consists of all binary strings together with the single string 2.

There are common shorthand notations which avoid cumbersome expressions for combinations of languages. For example, we write AB instead of $A \cdot B$,

$$010^*1^*0 \text{ instead of } \{01\} \cdot \{0\}^* \cdot \{1\}^* \cdot \{0\},$$

and

$$abb(ab)^* + (a+b)^*ba^* \text{ instead of } \{abb\} \cdot \{ab\}^* + \{a,b\}^* \cdot \{b\} \cdot \{a\}^*.$$

1

What Is a Turing Machine?

A Turing machine is ... the ultimate personal computer, since only pencil and paper are needed ... at the same time, it is as powerful as any real machine. ([34], p. 280)

We begin our study of computability by describing one of the earliest mathematical models of computation, one for which the underlying informal picture is especially easy to understand—the Turing machine.

In that picture (see Figure 1), a Turing machine consists of an infinite tape, and a read/write head connected to a control mechanism. The tape is divided into infinitely many cells, each of which contains a symbol from an alphabet called the tape alphabet; this alphabet includes the special symbol **B** to signify that a cell is blank (empty). The cells are scanned, one at a time, by the read/write head, which can move in both directions as long as it does not move off the tape (which would happen if, for example, the tape was bounded on the left and the read/write head moved left from the leftmost cell). At any given time, the machine (or, more properly, its control mechanism) will be in one of a finite number of possible states. The behaviour of the read/write head, and the change, if any, of the machine's state, are governed by the present state of the machine and by the symbol in the cell under scan.

The machine operates on words over an input alphabet which is a subset of the tape alphabet. The symbols forming such a word are written, in order, in consecutive cells from the left of the tape. When the machine enters a state, the read/write head reads the symbol in the cell against which it rests, and writes in that cell a symbol from the tape alphabet; it then moves one cell to the left, or one cell to the right, or not at all; after that, the machine enters its next state.

In this model there is no direct counterpart to the memory registers of a computer. However, information is stored in the sequence of states through which the machine passes. For example, if we want a Turing machine to transfer the content of a certain cell to the adjacent cell on the right, we "memorise" the symbol s read from the first cell by passing to a different state for each possible choice of s.

We now give a formal definition of some of these notions. Let X, Y be finite alphabets with $X \subset Y$, and **B** a distinguished **blank** element of

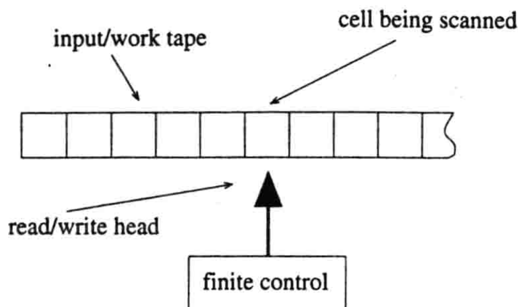


FIGURE 1. A Turing machine.

$Y \backslash X$. A **Turing machine** with **tape alphabet** Y and **input alphabet** X is a quadruple $\mathcal{M} \equiv (\mathcal{Q}, \delta, q_0, q_F)$ consisting of

- a finite set $\mathcal{Q}$ of **states**,
- a partial function $\delta : \mathcal{Q} \times Y \to \mathcal{Q} \times Y \times \{L, R, \Lambda\}$—the **state transition function**,
- a **start state** $q_0 \in \mathcal{Q}$, and
- a **halt state** $q_F \in \mathcal{Q}$,

where $\delta(q_F, y)$ is undefined for all y in Y.[1] We interpret the symbols L, R, and Λ as *left move, right move,* and *no move,* respectively.

We shall discuss examples of Turing machines later in the chapter. Our next task is to clarify our informal picture of the behaviour of a Turing machine.

In order to start a computation, the symbols of the input word

$$w \equiv x_1 \ldots x_N \in X^*$$

must be written in the leftmost N cells of the tape, and $\mathcal{M}$ must be in the state q_0, with the read/write head against the leftmost cell. If $\mathcal{M}$ reads the symbol y in the state q, it computes $(q', y', D) = \delta(q, y)$, provided this quantity is defined. It then writes y'; moves left if $D = L$, right if $D = R$, not at all if $D = \Lambda$; and passes to the state q'. If $\mathcal{M}$ reaches the state

[1]Strictly speaking, we have defined here a **deterministic Turing machine.** This should be contrasted with a nondeterministic one, in which there is a choice of several actions when the machine reads a given symbol in a given state. Since we shall not be concerned with nondeterministic Turing machines, we shall use the shorter phrase *Turing machine*, rather than *deterministic Turing machine*, throughout this book.

q_F, its activity stops and the final output of its computation is read from the symbols remaining on the tape. (Actually, we need to be more careful about characterising the moves, halting behaviour, and outputs of $\mathcal{M}$; we will return to this matter shortly.)

Suppose that at a given instant our Turing machine $\mathcal{M}$ is in the state q; that the symbols in the cells on the left of the read/write head form the string $u \in Y^*$; that the terms of a string $v \in Y^*$ lie in the cells at, and to the right of, the read/write head; and that all cells to the right of v are blank. Thus the leftmost cells of the tape contain the string uv, and all cells to the right of this are blank. Then the instantaneous configuration of the machine is fully described by the triple (u, q, v), and the state transitions of $\mathcal{M}$ can be described by the sequence of triples giving the configurations of $\mathcal{M}$ at successive instants of the computation.

In order to formalise these ideas, we introduce two intuitively computable functions from Y^* to Y :

$$\begin{aligned} \mathbf{lend}(v) &= \mathbf{B} \quad \text{if } v = \Lambda, \\ &= c \quad \text{if } v = cw, c \in Y, \text{ and } w \in Y^*, \end{aligned}$$

$$\begin{aligned} \mathbf{rend}(v) &= \mathbf{B} \quad \text{if } v = \Lambda, \\ &= c \quad \text{if } v = wc,\ c \in Y, \text{ and } w \in Y^*. \end{aligned}$$

(Thus if v is a nonempty string over Y, then $\mathbf{lend}(v)$ is the leftmost symbol, and $\mathbf{rend}(v)$ is the rightmost symbol, of v.) Next, we define a **configuration** of $\mathcal{M}$ to be a triple (u, q, v), where $u \in Y^*$, either $v = \Lambda$ or $v \in Y^*(Y \backslash \{\mathbf{B}\})$, and $q \in Q$.[2] We say that the configuration (u', q', v') is **reached in one step** from (u, q, v) if

$$\delta(q, \mathbf{lend}(v)) = (q', b, D) \in Q \times Y \times \{L, R, \Lambda\}$$

is defined (so, in particular, $q \neq q_F$), and if the following conditions obtain.

(i) If $D = L$, then $u = u'\mathbf{rend}(u)$ and

$$\begin{aligned} v' &= \Lambda && \text{if } b = \mathbf{B},\ \mathbf{rend}(u) = \mathbf{B}, \text{ and either } v = \Lambda \text{ or } v = \mathbf{lend}(v), \\ &= \mathbf{rend}(u) && \text{if } b = \mathbf{B},\ \mathbf{rend}(u) \neq \mathbf{B}, \text{ and either } v = \Lambda \text{ or } v = \mathbf{lend}(v), \\ &= \mathbf{rend}(u)bw && \text{if } w \in Y^*,\ v = \mathbf{lend}(v)w, \text{ and either } b \neq \mathbf{B} \text{ or } w \neq \Lambda. \end{aligned}$$

[2]As it stands, this definition does not completely capture our intuitive conception of a configuration, since it does not preclude the possibility of nonblank symbols lying to the right of the string v on the tape. However (see Exercise (1.2.2)), this situation does not arise when the configuration (u, q, v) is part of a computation, according to the strict notion of computation that we shall introduce shortly.

此为试读，需要完整PDF请访问：www.ertongbook.com