Machines, Languages, and Computation

Peter J. Denning
Jack B. Dennis
Joseph E. Qualitz

Machines, Languages, and Computation

Peter J. Denning
Purdue University

Jack B. Dennis
Massachusetts Institute of Technology

Joseph E. Qualitz
Artisan Industries, Inc.

Prentice-Hall, Inc., Englewood Cliffs, New Jersey 07632

Preface

Theoretical computer science has evolved from three disciplines: mathematics, engineering, and linguistics. The mathematical roots of computer science date from the 1930s when Turing's work exposed fundamental limits on mechanical computation. This work, an outgrowth of mathematical logic, predates stored computers by more than a decade. Turing's discovery was reinforced by the work of Church, Kleene, and Post on recursive functions and the formalization of mathematical logic. The origins in engineering began with Shannon's observation, in 1938, that the functions of relay switching networks could be represented in the symbolic notation of Boolean algebra. In the mid-1950s, Caldwell and Huffman extended this work to obtain a formal approach to sequential switching circuits, which evolved into the theory of finite-state machines. The linguistic contribution to computer science came in the late 1950s with Chomsky's characterization of formal grammars and languages. Interest in formal specification of programming languages led many computer scientists to study Chomsky's work and build it into an important theory of artificially constructed languages.

This book explores these three underlying themes of the theory of computation, and is intended for computer science undergraduates at the senior level—especially those planning to pursue graduate study. Several selections of material may be chosen for a one-semester course, according to the emphasis desired by the instructor. Alternatively, the book may be used for a comprehensive two-semester course at a less intensive pace, or for sophomores or juniors with a good mathematics background. (Suggested chapter selections are discussed on page xiii, under *Advice for Instructors.*)

Study of the theory of computation is an essential component of any computer science curriculum. For its further evolution, modern computer science depends on new methodologies for the construction of computers, systems, languages, and application programs so that we may be confident in their ability to perform as intended by their designers. The need for accurate descriptions of computers, languages, and programs, and for precise characterization of their behavior, has spurred the development of theoretical science. This includes work on the formal specification of programs, languages, and systems, and on methods of proving their correctness; on semantic formalisms that provide a sound foundation in mathematics for basing theories of program and system behavior; and on computational complexity—the study of the structure and space/time requirements of algorithms. Since entire books are devoted to these areas,† we devote the present volume to the fundamental knowledge—abstract machines and languages, and computability—essential to understanding and contributing to these developing areas of computer science.

This book had its genesis in an MIT course taught continuously since 1965, and has seen many years of evolution and refinement. It differs from major related texts‡ in several ways. First, we have emphasized the integrated development of the main results in each of the three major conceptual threads (machines, languages, computability), showing the relations and equivalences among them. Second, we appeal to the student's knowledge of programming by using program-like descriptions of control units and algorithms. Third, we have tried to develop the material in a way that has intuitive appeal. Though our arguments are rigorous, every concept is carefully developed and well illustrated through examples.

Content

Figure 1.4 shows the hierarchy of machines and languages studied in this book, and the chapters in which each topic is treated. The following paragraphs summarize the contents of each chapter.

Chapter 1 introduces abstract machines, and presents abstract languages as sets of strings over finite alphabets. Chapter 2 briefly reviews elements of set theory and logic, which provide concepts and notation relied on throughout the book; it also introduces terminology for strings of symbols and the important operations on strings and languages.

In Chapter 3, we begin the formal study of languages with the concept of

†Aho, Hopcroft, Ullman [1974]; Borodin & Munro [1975]; Braffort & Hirschberg [1967]; Stoy [1977]; Yeh [1977].

‡Arbib [1969;] Hennie [1968]; Hermes [1965]; Hopcroft & Ullman [1969]; Kain [1972]; Minsky [1967]; Stoll [1963].

representing an infinite set of sentences by a finite set of grammatical rules that generate the sentences of the language. The structural description of languages is illustrated by examples from English and the programming language Algol. Chomsky's hierarchy of grammars (and corresponding languages) is defined. Syntax trees are defined for context-free languages, and derivation diagrams for the more expressive language classes. Ambiguous derivations are defined.

Chapter 4 begins the study of abstract machines with a physically-motivated discussion of the finite automaton, a machine built from a finite number of elementary parts. The concept of "state" is introduced to formalize the description of a machine's internal configuration. Two common methods of assigning machine outputs (to transitions or to states of the machine) are shown to be equivalent. Equivalent machines and states are studied, and a procedure for reducing a finite-state machine to minimal canonical form is developed. The chapter treats briefly the characterization of a machine by equivalence classes of input sequences, grouped according to the states in which they leave the machine.

Chapter 5 combines concepts from Chapters 3 and 4 in the study of finite-state languages—sets of strings recognizable by finite state machines. Nondeterministic machines are introduced and are shown to be reducible to deterministic machines (usually at the cost of greater machine complexity). Methods of converting between Chomsky's linear grammars and nondeterministic machines are given. Kleene's regular expressions are introduced, and methods of transforming between regular expressions and finite-state machines are developed. The chapter ends by discussing closure properties, ambiguity, and decision problems for finite-state languages.

Chapter 6 studies the limitations of finite-state machines. It exhibits languages for which there exist no finite-state accepters. It characterizes the limitations of finite-state accepters, generators, and transducers. It characterizes the ultimately periodic behavior of finite-state machines, and proves, based on this behavior, that certain arithmetic operations (for example, multiplication) are beyond the capabilities of finite automata.

Chapter 7 introduces tape automata. In a first attempt to define a class of automata more powerful than the finite-state machines, we study tape automata with read-only input and no external writeable stores. It is shown that, even with two-way motion on their input tapes, these automata have no more computing power than finite-state machines; we conclude that additional storage is required to increase their computational power.

Chapter 8 studies pushdown automata, a class of tape automata having a restricted, but unbounded, external store: items can be retrieved from the store only in reverse order of entry. The unbounded store endows these machines with more power than finite automata—for example, they can recognize languages that are not finite state—but we show that the restriction on their store access method limits the ultimate power of these machines also. Using a concept of traverse sets—sets of input strings that lead a machine to the same control state and store configuration—we show how to transform between context-free languages and nondeterministic pushdown accepters. This equivalence is exploited to develop the closure properties of the context-free languages. We discover languages which cannot be recognized by any pushdown machine; we also discover languages which can be recognized by pushdown automata only if nondeterministic behavior is permitted. (That the deterministic languages are a proper subset of all context-free languages implies that, in practice, syntax analyzers for some context-free languages require backtracking, and thus may be slow; this implication is studied in Chapter 10.) Finally, we consider the effect of limiting the number of distinct symbols in an automaton's tape alphabet. Two symbols are shown to be sufficient for general behavior, a single symbol is shown to define a class of counting automata intermediate in power between the finite-state machines and the pushdown automata.

Chapter 9 presents properties of context-free languages which derive readily from a study of their grammars. After showing how to eliminate needless productions from a grammar, we prove that these grammars can be reduced to either of two canonical forms: the Chomsky normal form and the Greibach standard form. These forms are frequently useful for simplifying proofs and, sometimes, syntax analyzers for context-free languages. Two important characterizations of context-free languages are proved: the pumping lemma shows that, when a sufficiently long string is in a context-free language, an infinity of related strings must also be there; the self-embedding theorem characterizes the possible substring matchings in context-free languages. These theorems are used to prove that certain languages are not context free.

Chapter 10 deals with syntax analysis of context-free languages. Most modern compilers use parsing techniques based on context-free languages; this chapter develops the most important methods. Top-down and bottom-up analyses are defined and compared. Two forms of deterministic bottom-up analysis are studied: (simple mixed strategy) precedence analysis, and $LR(k)$ analysis. In each case, we show how to construct an appropriate parser and show that the corresponding grammars generate all deterministic context-free languages.

In Chapter 11 we study Turing machines, the simplest models of general mechanical computation. A variety of possible Turing machines are shown equivalent to a simple form of Turing machine which has unrestricted access to a single, singly-infinite storage tape. (The linear-bounded automaton, a machine similar to the Turing machine but with a storage tape whose length is bounded linearly by the length of its input, is discussed briefly.) A variety of Turing machine programs for simple operations are developed, and are used as procedures in a universal machine capable of simulating any other Turing machine. We state and explain Turing's thesis: the solution procedure for any problem that can be solved effectively can be programmed on a Turing machine. The universal machine partly supports this claim. Additional support is provided by the equivalence of Turing machines to two other, seemingly unrelated, models of effective computation: recursive functions and Post string manipulation systems. These equivalences are demonstrated in Chapters 13 and 14. Finally, we define the concepts of enumeration, recursive sets, and recursively enumerable sets.

Chapter 12 deals with unsolvable problems, well-defined classes of problems for which, by Turing's thesis, there exist no effective solution procedures. These include program-termination problems, word problems, and a variety of decision problems concerning context-free languages. The existence of mechanically unsolvable problems is one of the most fundamental results of the theory of computation.

In Chapter 13 we study recursive functions, a class of number-theoretic functions used to model computation. We study three classes of recursive functions: the primitive recursive functions, the μ-recursive functions, and the multiple-recurisve functions. The latter two classes are shown to be equivalent, and each is shown equivalent to the Turing machines. Specifically, we show how to construct a Turing machine that evaluates a given multiple-recursive function; we show how to construct a μ-recursive function that simulates a given Turing machine; and we show that the class of multiple-recursive functions contains the class of μ-recursive functions.

Chapter 14 deals with Post string manipulation systems, a third model of computation. Post systems are a formalization of the processes by which humans solve problems symbolically, as in mathematical logic. We show how to represent a few common algorithms with these systems, and then show how any Turing machine may be encoded as a Post system and vice versa—thus demonstrating the equivalence between Turing machines and Post systems, and completing our study of the evidence supporting Turing's thesis.

Advice for Instructors

Covering the entire content of this book in a one-semester course is possible for students with a strong background in basic set theory and logic, and experience in proving theorems; however, a fast pace is necessary. Many teachers will likely prefer either to cover the material in two semesters or to teach a one-semester course emphasizing some portion of the material.

Chapter 2 reviews the essential mathematical background and introduces the operations on strings used extensively throughout this book; all but Section 2.7 may be skipped for students with prior study in set theory and logic. Other parts of the book that are of secondary importance and may be skipped without loss of continuity include Sections 6.4, 6.5, 7.3, 8.6.2, and 8.7. Chapter 9 develops properties of context-free languages and grammars useful for the study of parsing methods in Chapter 10. (The theorems in Section 9.3, although useful for establishing the level of a language in the Chomsky hierarchy, are not used later in the book.) Chapter 10 itself may be omitted if a treatment of parsing algorithms is not desired.

Sections 13.5, 13.6, and 14.2 contain the detailed developments that establish simulation results in support of Church's thesis. They are included for completeness, and may be treated briefly if time is short.

For a one-semester course on language concepts (in preparation, for example, for the study of program language implementation), Chapters 3, 5, and 8 through 10 should be emphasized. For a one-semester course based on Church's thesis, Chapters 11 through 14 should be covered in depth, using selected material from Chapters 3 through 8 as background.

Problems and References

Each chapter ends with a selection of Problems. Many of these are designed as exercises to give the student confidence in his understanding of the material and practice in applying procedures developed in the chapter. Others call for the proof of results not proved in the text, or extend, in some way, material developed in the text. These often call for some creativity on the part of the student, and are marked in boldface. Problems marked with an asterisk are more difficult and are included to challenge the student.

A bibliography of books and papers cited in the text appears on pages 584-592. At the end of each chapter, we include remarks on the origin and historical development of the major concepts treated in the chapter. Similar remarks are included in some of the Problems.

Acknowledgements

We are grateful to Marsha Baker and Anne Rubin, whose typing of (countably) many versions of the manuscript nurtured it from a small set of lecture notes in 1967 to a manuscript four times larger in 1977.

We are indebted to our course instructors over the years who, in their quest for perfection, reproved us for every blemish and error thay could find; on their account, we have reproved many theorems. These people are: John DeTreville, Kennith Dritz, Peter Elias, Irene Greif, D. Austin Henderson, Carl Hewitt, Suhas Patil, and Richard Spann.

Seven senior computer scientists were especially influential in guiding us to the important concepts and showing us, all too frequently, simpler ways of

doing things. They are: Manuel Blum, Frederick Hennie, Richard Kain, David Kuck, David Martin, Robert McNaughton, and Albert Meyer. Of these seven, two deserve special note. Professor David Kuck, now of the University of Illinois, shared with Jack Dennis the responsibility for this course when it was instituted in 1965; his influence on the structure of the course and the book was considerable. Professor Robert McNaughton, now of Rensselaer Polytechnic Institute, was a walking encyclopedia of automata theory and formal linguistics; he generously gave of his time and knowledge during the formative stages of this work, and led us to an early appreciation of the research in formal linguistics at Harvard and MIT.

We cannot omit our admiration for Karl Karlstrom, our humble and faithful servant as Senior Editor at Prentice-Hall. His interest in our work began in 1967, when Denning and Dennis agreed to deliver a manuscript the following year. Qualitz, who had instructed the course at MIT several times, rescued the project from oblivion by completing the manuscript and making needed revisions to the work which had languished as the other authors took up other interests. Karl, whose patience and encouragement did not sag for a decade, tells his friends that he knew all along: his secretary meant to type "78", not "68", in the delivery date. Thanks, Karl, for your confidence and support.

PETER J. DENNING
JACK B. DENNIS
JOSEPH E. QUALITZ

About the Authors

Peter J. Denning is Professor of Computer Science at Purdue University where his primary research interests are modeling and analysis of computer performance, design of operating systems, memory management and program behavior, data security and protection, secure data communication, fault tolerant software, and parallel computation. His work has led to many publications including the book Operating Systems Theory *with E. G. Coffman, Jr. He has served the Association for Computing Machinery (ACM) in many official positions, and is very active in editorial work, including serving as Editor-in-Chief of ACM's* Computing Surveys. *Peter Denning was born in New York City, and grew up in Darien, Connecticut. He received the Ph.D degree from MIT in 1968, and joined the Purdue faculty after spending four years as Assistant Professor of Electrical Engineering at Princeton University. He has received two best paper awards, and a teaching award from Princeton. He is an inveterate jogger and wine connoisseur.*

Jack B. Dennis is Professor of Computer Science and Engineering at MIT where he leads a research group in work on advanced concepts of computer system architecture in the MIT Laboratory for Computer Science. Jack Dennis earned his doctorate from MIT in 1958 for work relating mathematical programming and the theory of electric circuits. Since then he has been involved in developing new course offerings in basic computer science at MIT, in working with a number of successful doctoral research students, and in organizing professional conferences. He was elected Fellow of the Institute of Electrical and Electronic Engineers for his contributions to the design of computer memory systems. He is a New Jersey native and received his early education in Darien, Connecticut. He now resides in Belmont, Massachusetts and enjoys tennis, hiking, and singing with choral groups.

Joseph E. Qualitz is a native of Waltham, Massachusetts. He received an SB and SM in Electrical Engineering from MIT in 1972, and a Ph.D in Computer Science in 1975. He served as Instructor in the MIT Department of Electrical Engineering from 1972 to 1975, and in 1973 received a teaching award as outstanding instructor. Dr. Qualitz is currently Chief Computer Engineer of Artisan Industries, Inc., of Waltham, Mass., where he is engaged in the design and implementation of microcomputer development and support systems.

Table of Major Theorems

Chapter 4: Finite-State Machines

Theorem 4.1: For each state-assigned machine there exists a similar transition-assigned machine. Conversely, for each transition-assigned machine there exists a similar state-assigned machine.

Theorem 4.4: There is an effective procedure for partitioning the states of a finite-state machine into blocks of equivalent states.

Theorem 4.5: The state graphs of reduced, connected finite-state machines are isomorphic if and only if the machines are equivalent.

Chapter 5: Finite-State Languages

Theorem 5.1: For each finite-state accepter $\mathbf{M}_n$, one can construct a deterministic finite-state accepter $\mathbf{M}_d$ such that $\mathbf{L}(\mathbf{M}_d) = \mathbf{L}(\mathbf{M}_n)$.

Theorem 5.2: For any finite-state accepter **M**, one can construct a right-linear grammer **G** such that $\mathbf{L}(\mathbf{G}) = \mathbf{L}(\mathbf{M})$.

Theorem 5.3: For any right-linear grammar **G**, one can construct a finite-state accepter **M** such that $\mathbf{L}(\mathbf{M}) = \mathbf{L}(\mathbf{G})$.

Theorem 5.5: Each finite-state accepter recognizes a language that can be described by some regular expression.

Theorem 5.6: For any regular expression α, one can construct a finite-state accepter $\mathbf{M}$ such that $\mathbf{L(M)} = \mathbf{R}$, where $\mathbf{R}$ is the set described by α.

Theorem 5.7: The class of finite-state languages is closed under the operations of set union, intersection, complementation, difference, concatenation, closure, and reversal.

Theorem 5.8: There is a finite procedure for deciding whether an arbitrary right-linear grammar is ambiguous, and, if so, for finding an ambiguous sentence generated by the grammar.

Theorem 5.9: For any regular grammar $\mathbf{G}$, it is possible to construct an unambiguous grammar $\mathbf{G}'$ such that $\mathbf{L(G')} = \mathbf{L(G)}$.

Theorem 5.10: Let $\mathbf{L}_1$ and $\mathbf{L}_2$ be arbitrary finite-state languages, and let $\mathbf{G}$ be an arbitrary regular grammar. Then it is decidable whether

1. $\mathbf{L}_1 = \mathbf{L}_2$.
2. $\mathbf{L}_1 = \varnothing$.
3. $\mathbf{L}_1$ is finite; $\mathbf{L}_1$ is infinite.
4. $\mathbf{L}_1 \cap \mathbf{L}_2 = \varnothing$.
5. $\mathbf{L}_1 \subseteq \mathbf{L}_2$.
6. $\mathbf{G}$ is ambiguous.

Chapter 6: Limitations of Finite Automata

Theorem 6.1 (Finite-State Language Theorem): Let $\mathbf{L}$ be a regular set, and suppose that there is an integer $p \geq 0$ such that

$$\mathbf{X} = \{\alpha_1(\alpha_2)^k\alpha_3(\alpha_4)^k\alpha_5 \mid k \geq p\}$$

is contained in $\mathbf{L}$, where $\alpha_1, \ldots, \alpha_5$ are strings with α_2 and α_4 nonempty. Then there exist strings $\beta_1, \ldots, \beta_5$ such that

$$\mathbf{Y} = \beta_1(\beta_2)^*\beta_3(\beta_4)^*\beta_5$$

is a subset of $\mathbf{L}$, and

$$\beta_1 \in \alpha_1\alpha_2^*$$

$$\beta_2 \in \alpha_2^* - \lambda$$

$$\beta_3 \in \alpha_2^*\alpha_3\alpha_4^*$$

$$\beta_4 \in \alpha_4^* - \lambda$$

$$\beta_5 \in \alpha_4^*\alpha_5$$

Theorem 6.2: A language is regular if and only if it is generated by some (nondeterministic) finite-state generator.

Theorem 6.3: The transduction of a regular set by a finite-state transducer is a regular set. That is, the class of regular sets on a given alphabet is closed under finite-state transducer mappings.

Theorem 6.4: (1) Let $\mathbf{M}$ be an n-state deterministic finite-state generator. Then $\mathbf{M}$ generates an ultimately periodic language $\mathbf{L(M)} = \mathbf{T} \cup \tau\rho^*\mathbf{P}$, where $|\tau| + |\rho|$

$\leq n$. (2) Conversely, each ultimately periodic language $\mathbf{L} = \mathbf{T} \cup \tau\rho^*\mathbf{P}$ is generated by some deterministic finite-state generator with no more than $|\tau\rho|$ states; moreover, if τ and ρ are respectively the basic transient and basic period of **L**, then no deterministic generator for **L** has fewer than $|\tau\rho|$ states.

Theorem 6.5: Let $\mathbf{X} \subseteq \mathbf{S}^*$ be an ultimately periodic language, and let $\mathbf{M}_t$ be a deterministic finite-state transducer with input alphabet **S**. Then the transduction of **X** by $\mathbf{M}_t$ is an ultimately periodic language **Y**. Furthermore, if τ and ρ are the basic transient and basic period of the input **X**, and α and β are the basic transient and basic period of the output **Y**, then

$$|\alpha\beta| \leq |\tau| + n_t|\rho|$$

where n_t is the number of states in $\mathbf{M}_t$.

Chapter 7: Tape Automata

Theorem 7.1: The class of regular sets is closed under transduction by a generalized sequential machine.

Theorem 7.2: The class of regular sets is closed under inverse transduction by a generalized sequential machine.

Theorem 7.3: From any deterministic two-way accepter **M**, one can construct a finite-state accepter **M′** such that **L(M′)** = **L(M)**.

Chapter 8: Pushdown Automata

Theorem 8.1: For any context-free grammar **G**, one can construct a pushdown accepter **M** such that **L(M)** = **L(G)**. Moreover, the accepting move sequences in **M** are in one-to-one correspondence with leftmost derivations in **G**.

Theorem 8.2: For any pushdown accepter **M**, one can construct a context-free grammar **G** such that **L(G)** = **L(M)**.

Theorem 8.4: The class of context-free languages is closed under the operations union, concatenation, closure, reversal, transduction by a generalized sequential machine, and intersection and difference with a regular set.

Theorem 8.5: The deterministic context-free languages are a proper subclass of all context-free languages. In particular, the language $\mathbf{L}_{nd} = \{a^k b^m \mid m = k$ or $m = 2k, k \geq 1\}$ is context free but not deterministic.

Theorem 8.6: The class of deterministic context-free languages is closed under the operations complement, intersection with a regular set, and difference with a regular set. It is not closed under the operations union, intersection, concatenation, reversal, set closure, or (deterministic) transduction by a finite-state machine.

Theorem 8.7: Counting accepters are intermediate in power between finite-state accepters and pushdown accepters. In particular, there is a counting accepter that recognizes the parenthesis language $\mathbf{L}_p$, but no counting accepter that recognizes the double parenthesis language $\mathbf{L}_{dp}$.

Theorem 8.8: The class of deterministic context-free languages is a proper subclass of the unambiguous context-free languages.

Chapter 9: Context-Free Languages

Theorem 9.1 (Emptiness Test): For any context-free grammar **G**, one can decide whether $\mathbf{L}(\mathbf{G}, A) = \varnothing$ for any nonterminal symbol A in **G**. In particular, one can decide whether the grammar generates any strings at all [that is, whether $\mathbf{L}(\mathbf{G}) = \mathbf{L}(\mathbf{G}, \Sigma) = \varnothing$].

Theorem 9.3 (Normal-Form Theorem): From any context-free grammar, one can construct a strongly equivalent grammar in normal form.

Theorem 9.4 (Standard Form Theorem): From any context-free grammar, one can construct a strongly equivalent grammar in standard form.

Theorem 9.5 (Structure Theorem): Let $\mathbf{G} = (\mathbf{N}, \mathbf{T}, \mathbf{P}, \Sigma)$ be a well-formed context-free grammar. For any A in $\mathbf{N} \cup \{\Sigma\}$, $\mathbf{L}(\mathbf{G}, A)$ is infinite if and only if **G** permits the following derivations for some nonterminal B:

1. $A \overset{*}{\Longrightarrow} \alpha B \beta, \; \alpha, \beta \in \mathbf{T}^*$.
2. $B \overset{*}{\Longrightarrow} \varphi B \psi, \; \varphi\psi \in \mathbf{T}^* - \lambda$.
3. $B \overset{*}{\Longrightarrow} \sigma, \qquad \sigma \in \mathbf{T}^* - \lambda$.

Theorem 9.6: For any context-free grammar **G**, one can decide whether $\mathbf{L}(\mathbf{G})$ is finite or infinite.

Theorem 9.7 (Pumping Lemma): If **L** is a context-free language, there exists a positive integer p with the following properties: whenever ω is in **L** and $|\omega| > p$, there exist strings $\alpha, \varphi, \sigma, \psi$, and β, with $\varphi\psi$ and σ nonempty and $|\varphi\sigma\psi| \leq p$, such that $\omega = \alpha\varphi\sigma\psi\beta$ and $\alpha\varphi^k\sigma\psi^k\beta$ is in **L** for all $k \geq 0$.

Theorem 9.8: The class of context-free languages is properly contained in the class of context-sensitive languages. In particular, $\mathbf{L}_{dm} = \{a^n b^n c^n \mid n \geq 1\}$ is context sensitive but not context free.

Theorem 9.9 (Self-embedding Theorem): A context-free language is nonregular if and only if every grammar generating the language is self-embedding.

Chapter 10: Syntax Analysis

Theorem 10.2: A context-free language is deterministic if and only if it is generated by a generalized precedence grammar.

Theorem 10.3: Given any context-free grammar **G**, one can determine, for any $k \geq 0$, whether **G** is an *LR*(k) grammar.

Theorem 10.4: (1) Every deterministic context-free language is generated by some $LR(1)$ grammar. (2) Every $LR(k)$ language is deterministic.

Chapter 11: Turing Machines

Theorem 11.1: The domain of the relation realized by any nondeterministic Turing machine is a Turing-recognizable language.

Theorem 11.2: The class of Turing-recognizable languages properly includes the class of context-free languages.

Theorem 11.3: One can construct a universal Turing machine **U** that realizes a function

$$f_{\mathrm{U}}: \mathbf{A}^* \longrightarrow \mathbf{A}^*$$

such that for any Turing machine **M** with tape alphabet **T** and any string $\omega \in \mathbf{T}^*$, we have

$$f_{\mathrm{U}}(D(\mathbf{M}) \# T(\omega)) = \begin{cases} C(\alpha) & \text{if } \mathbf{M} \text{ has a halted computation } \alpha_0 \Longrightarrow \alpha \\ \text{undefined} & \text{otherwise} \end{cases}$$

where $\alpha_0 = (\lambda, q_I, \#, \omega)$, and $D(\mathbf{M})$, $T(\omega)$, and $C(\alpha)$ are specific encodings of $\mathbf{M}$, ω, and α in the finite alphabet **A**.

Theorem 11.4: A set $\mathbf{X} \subseteq \mathbf{N}$ is Turing enumerable if and only if **X** is the domain of a Turing computable function.

Theorem 11.5: A set $\mathbf{X} \subseteq \mathbf{N}$ is Turing semidecidable if and only if it is Turing enumerable.

Theorem 11.6: A set is Turing recognizable if and only if it is Turing enumerable.

Theorem 11.7: There exists a Turing computable function

$$W: \mathbf{N} \times \mathbf{N} \longrightarrow \mathbf{N}$$

such that for all z, x, y

$$W(z, x) = y$$

if and only if machine $\mathbf{M}_z$ in an enumeration of Turing machines computes

$$f_z(x) = y$$

Chapter 12: Unsolvable Problems

Theorem 12.1: The halting problem for Turing machines is unsolvable.

Theorem 12.2: The busy-beaver function is noncomputable.

Theorem 12.3: The word problem for type 0 grammars is unsolvable.

Theorem 12.4: The correspondence problem is unsolvable.

Theorem 12.5: The problem of deciding, for arbitrary context-free languages **L** and **L′**, whether $\mathbf{L} \cap \mathbf{L}'$ is empty (or infinite) is unsolvable.

Theorem 12.6: The problem of deciding, for an arbitrary context-free language **L**, whether $\mathbf{L}^c$ is empty (or infinite) is unsolvable.

Theorem 12.7: There is no effective procedure for deciding whether a given context-free language is regular.

Theorem 12.8: There is no effective procedure for determining whether a given context-free language is deterministic.

Theorem 12.9: There is no effective procedure for deciding, given any context-free grammar **G**, whether **G** is ambiguous.

Theorem 12.10: The problem of deciding whether $\mathbf{L(G)} = \mathbf{L(G')}$ for arbitrary context-free grammars **G** and **G′** is unsolvable.

Chapter 13: Recursive Functions

Theorem 13.1: The μ-recursive functions, the multiple-recursive functions, and the Turing-computable functions are equivalent classes of functions.

Theorem 13.2: Let $\mathbf{A} \subseteq \mathbf{N}$ and $\mathbf{A}^c = \mathbf{N} - \mathbf{A}$. Then:
(1) **A** is recursive if and only if $\mathbf{A}^c$ is recursive.
(2) **A** is recursive only if **A** is r.e.
(3) **A** is recursive just if both **A** and $\mathbf{A}^c$ are r.e.

Theorem 13.3: A set **A** is recursively enumerable if and only if **A** is the domain of a Turing-computable function. Equivalently, **A** is recursively enumerable if and only if **A** is the domain of a (partial) recursive function.

Theorem 13.4: The class of total recursive functions cannot be effectively enumerated.

Theorem 13.5: No recursively enumerable class of recursive sets contains every recursive set.

Chapter 14: Post Systems

Theorem 14.1: Post systems and Turing machines are equivalent representations of effective computability. That is, the deductions of a given Post system can be represented as the computations of some Turing machine, and the computations of a given Turing machine can be represented as the deductions of some Post system.

Contents