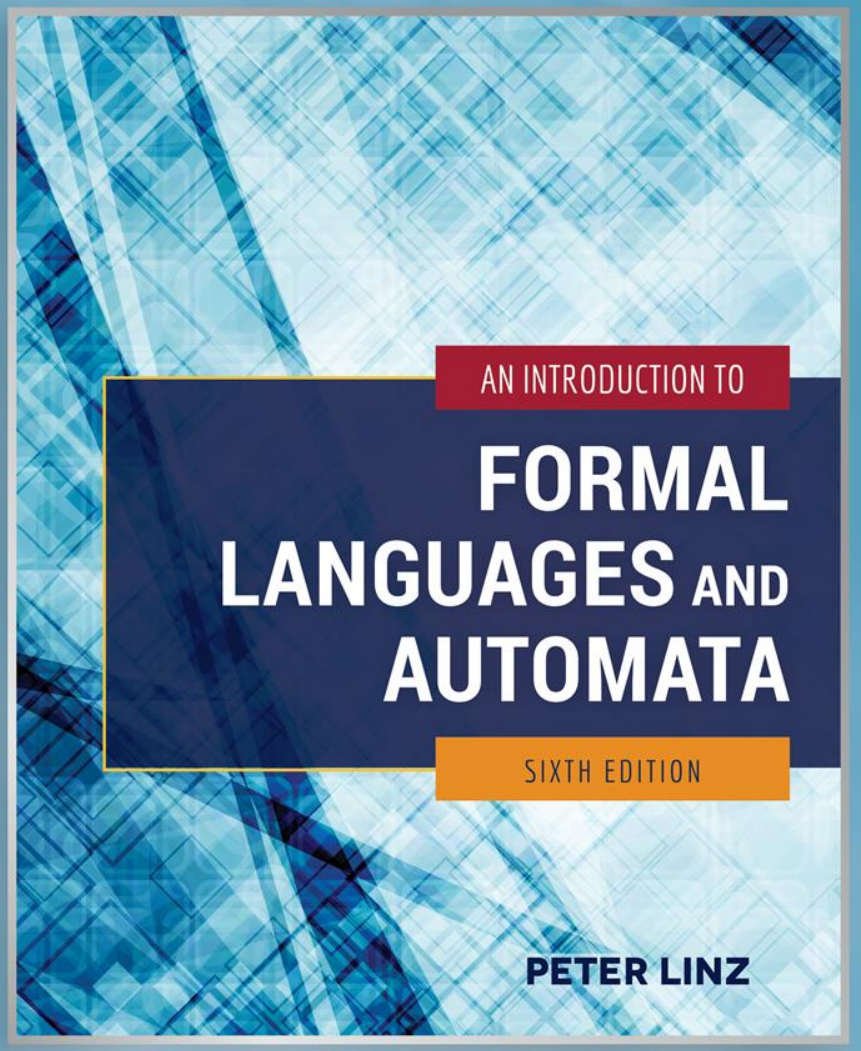
Chapter 13

OTHER MODELS OF COMPUTATION



Learning Objectives

At the conclusion of the chapter, the student will be able to:

- Define the basic functions and the operations used to build more complex functions
- Construct arithmetic operations using basic functions and operations
- Given a simple function, show that it is primitive recursive
- Evaluate Ackermann's function for sample arguments
- Define the concept of μ-recursive functions
- Describe the restrictions and characteristics of Post systems, matrix grammars, Markov algorithms, and L-systems
- Derive sample sentences generated by Post systems, matrix grammars, Markov algorithms, and L-systems

Basic Functions

- It is important to specify a formal notation for the definition of mathematical functions
- In all cases, the domain is either I (the set of all nonnegative integers) or the Cartesian product I x I, while the range is I
- The basic functions are
  - The **zero function** z(x) = 0, for all x $\in I$
  - The **successor function** s(x) = x + 1, for all x $\in I$
  - The **projector functions**

    $p_k (x_1, x_2) = x_k$, for k = 1, 2

Building Complex Functions

- There are two ways of building complex functions from the primitive functions: composition and primitive recursion
- **Composition:** given defined functions g_1, g_2, and h,

$$f(x, y) = h(g_1(x, y), g_2(x, y))$$

- **Primitive recursion:** given defined functions g_1, g_2, and h,

$$f(x, 0) = g_1(x)$$

$$f(x, y + 1) = h(g_2(x, y), f(x, y))$$

Constructing Arithmetic Operations

- Examples 13.1 and 13.2 show that arithmetic operations can be constructed from the basic functions
- Addition of integers x and y:

$$add\ (x, 0) = x$$

$$add\ (x, y + 1) = add\ (x, y) + 1$$

- Multiplication of integers x and y:

$$mult\ (x, 0) = x$$

$$mult\ (x, y + 1) = add\ (x, mult(x, y))$$

Primitive Recursive Functions

- A function is *primitive recursive* if and only if it can be constructed from the basic functions z, s, p_k, as well as composition and primitive recursion
- While most common functions are primitive recursive, there are functions from I to I which are not primitive recursive, as established by Theorem 13.1
- Theorem 13.2 goes further: there are total, computable functions from I to I which are not primitive recursive
- The nonconstructive proofs that confirm these results are based on the concept of *diagonalization*

Ackermann's Function

- An example of a total computable function from I to I which is not primitive recursive is known as Ackermann's function
- Ackermann's function is defined by

$$A(0, y) = y + 1$$
$$A(x, 0) = A(x - 1, 1)$$
$$A(x, y + 1) = A(x - 1, A(x, y))$$

- Ackermann's function grows rapidly, as shown below

$$\begin{aligned} A(3, 1) &= A(2, A(3, 0))) \\ &= A(2, A(2, 1))) \\ &= A(2, A(1, A(2, 0))) \\ &= A(2, A(1, A(1, 1))) \\ &= A(2, A(1, A(0, A(1, 0)))) \\ &= A(2, A(1, A(0, A(0, 1)))) \\ &= \ldots \end{aligned}$$

μ Recursive Functions

- Recursive functions can be extended by the introduction of the μ or *minimalization* operator
- Assuming that g is a total function, the *minimalization* operator is defined by

  $\mu y(g(x, y))$ = smallest y such that g(x, y) is 0

- As shown in Example 13.4, if g(x, y) = x + y – 3,

$$\mu y(g(x, y)) \quad \begin{cases} = 3 - x, & \text{for } x \leq 3 \\ = \text{undefined}, & \text{for } x > 3 \end{cases}$$

- A function is μ-recursive if it can be constructed from the basic functions by a sequence of applications of the μ-operator as well as composition and primitive recursion
- By Theorem 13.2, a function is μ-recursive if and only if it is computable

Post Systems

- A *Post system* is similar to an unrestricted grammar, but there are significant restrictions in the way the productions are applied
- A Post system Π is defined by
  - A finite set of constants C, consisting of two disjoint sets: C_N, called the *nonterminal constants*, and C_T, called the *terminal constants*
  - A finite set of *variables* V
  - A finite set from C* called the *axioms*
  - A finite set of *productions* P
- Each production is of the form

$$x_1V_1x_2 \ldots V_nx_{n+1} \rightarrow y_1W_1y_2 \ldots W_my_{m+1}$$

where $x_i, y_i \in C^*$, and any variable V_i can appear at most once on the left, and each variable W_i on the right must also appear on the left

The Language Generated by a Post System

- The language generated by a Post system Π = (C, V, A, P) is the set of all strings w in C_T^* generated by applying a sequence of productions that start with some w_0 in A
- As shown in Example 13.5, consider the Post system with

  $C_T = \{a, b\}$, $C_N = \varnothing$, $V = \{ V_1 \}$, $A = \{ \lambda \}$,

  and production $V_1 \rightarrow aV_1b$
- This system allows the derivation

  $\lambda \Rightarrow ab \Rightarrow aabb$
- It can be concluded that the language generated is $\{a^n b^n : n \geq 0\}$
- As stated in Theorem 13.6, a language L is recursively enumerable if and only if there exists some Post system that generates L

A Post System for Generating Identities of Integer Additions

- As shown in Example 13.6, the Post system below is designed to generate the set of all identities of integer additions using unary notation

$$C_T = \{ 1, +, = \},$$
$$C_N = \varnothing,$$
$$V = \{ V_1, V_2, V_3 \},$$
$$A = \{ 1 + 1 = 11 \},$$

  with productions

$$V_1 + V_2 = V_3 \rightarrow V_1 1 + V_2 = V_3 1$$
$$V_1 + V_2 = V_3 \rightarrow V_1 + V_2 1 = V_3 1$$

- This system allows the derivation

$$1 + 1 = 11 \Rightarrow 11 + 1 = 111 \Rightarrow 11 + 11 = 1111$$

Matrix Grammars

- The grammars previously studied are known as *phrase-structure grammars*
- Matrix grammars differ from phrase-structure grammars in how the productions can be applied
- For matrix grammars, the set of productions consists of subsets P_1, P_2, …, P_n, each of which is an ordered sequence of the form

$$x_1 \rightarrow y_1, x_2 \rightarrow y_2, \ldots$$

- Whenever the first production of some set P_i is applied, we must apply the second one, then the third one, etc.
- The first production in the set P_i can only be applied if all other productions in this set can also be applied

Languages Generated by Matrix Grammars

- As shown in Example 13.7, consider the matrix grammar

  $P_1: S \rightarrow S_1S_2$

  $P_2: S_1 \rightarrow aS_1, S_2 \rightarrow bS_2c$

  $P_3: S_1 \rightarrow \lambda, S_2 \rightarrow \lambda$

- This system allows the derivation

  $S \Rightarrow S_1S_2 \Rightarrow aS_1bS_2c \Rightarrow aaS_1bbS_2cc \Rightarrow aabbcc$

- The language generated is $\{a^nb^nc^n: n \geq 0\}$
- Phrase-structure grammars are a special case of matrix grammars in which each P_i contains exactly one production
- Although they are equivalent in power to phrase-structure grammars, matrix grammar allow for simpler solutions when generating complex languages

Markov Algorithms

- A *Markov algorithm* is a system whose productions of the form $x \rightarrow y$ are considered ordered:
  - In a derivation, the first applicable production must be used, and
  - The leftmost occurrence of the substring x must be replaced by y
- Some of the productions may be identified as terminal productions (followed by a period)
- A derivation starts with some string w in Σ and continues either
  - until a terminal production is used, or
  - until there are no applicable productions
- In a sense, Markov algorithms are grammars working backward: a string of terminals w is accepted if a derivation that starts with w generates the empty string

Sample Markov Algorithms

- As shown in Example 13.8, consider the Markov algorithm with $\Sigma = T = \{ a, b \}$, and productions

  $ab \rightarrow \lambda,$

  $ba \rightarrow \lambda.$
- Since every step in the derivation eliminates the substring ab or ba, the language generated consists of strings in Σ^* with an equal number of a's and b's
- As shown in Example 13.9, consider the Markov algorithm with $\Sigma = T = \{ a, b \}$, and productions

  $ab \rightarrow S,$

  $aSb \rightarrow S,$

  $S \rightarrow \lambda.$
- A sample derivation is: $aabb \Rightarrow aSb \Rightarrow S \Rightarrow \lambda$
- It can be shown that the language generated is $\{a^n b^n : n \geq 0\}$
- As stated in Theorem 13.7, a language L is recursively enumerable if and only if there exists a Markov algorithm that generates L

L-Systems

- *L-systems* are parallel rewriting systems used to model the growth pattern of certain organisms
- In an L-system, every symbol must be rewritten at each step of a derivation
- As shown in Example 13.10, consider the L-system with $\Sigma = \{ a \}$ and a single production $a \rightarrow aa$
- A sample derivation is: $a \Rightarrow aa \Rightarrow aaaa \Rightarrow aaaaaaaa$
- It is not difficult to conclude that the language generated is $\{ a^m : m = 2^n, n \geq 0 \}$
- An extension of the L-system definition produces a mechanism that servers as another general model of computation