Chapter 13

Other Models of Computation

Although Turing machines are the most general models of computation we can construct, they are not the only ones. At various times, other models have been proposed, some of which at first glance seemed to be radically different from Turing machines. Eventually, however, all the models were found to be equivalent. Much of the pioneering work in this area was done in the period between 1930 and 1940 and a number of mathematicians, A. M. Turing among them, contributed to it. The results that were found shed light not only on the concept of a mechanical computation, but on mathematics as a whole.

Turing's work was published in 1936. No commercial computers were available at that time. In fact, the whole idea had been considered only in a very peripheral way. Although Turing's ideas eventually became very important in computer science, his original goal was not to provide a foundation for the study of digital computers. To understand what Turing was trying to do, we must briefly look at the state of mathematics at that time.

With the discovery of differential and integral calculus by Newton and Leibniz in the seventeenth and eighteenth centuries, interest in mathematics increased and the discipline entered an era of explosive growth. A number of different areas were studied, and significant advances were made in almost all of them. By the end of the nineteenth century, the body of mathematical knowledge had become quite large. Mathematicians also had become sufficiently sophisticated to recognize that some logical difficulties had arisen that required a more careful approach. This led to a concern with rigor in reasoning and a consequent examination of the foundations of mathematical knowledge in the process. To see why this was necessary, consider what is involved in a typical proof in just about every book and paper dealing with mathematical subjects. A sequence of plausible claims is made, interspersed with phrases like "it can be seen easily" and "it follows from this." Such phrases are conventional, and what one means by them is that, if challenged to do so, one could give more detailed reasoning. Of course, this is very dangerous, since it is possible to overlook things, use faulty hidden assumptions, or make wrong inferences. Whenever we see arguments like this, we cannot help but wonder if the proof we are given is indeed correct. Often there is no way of telling, and long and involved proofs have been published and found erroneous only after a considerable amount of time. Because of practical limitations, however, this type of reasoning is accepted by most mathematicians. The arguments throw light on the subject and at least increase our confidence that the result is true. But to those demanding complete reliability, they are unacceptable.

One alternative to such "sloppy" mathematics is to formalize as far as possible. We start with a set of assumed givens, called **axioms**, and precisely defined rules for logical inference and deduction. The rules are used in a sequence of steps, each of which takes us from one proven fact to another. The rules must be such that the correctness of their application can be checked in a routine and completely mechanical way. A proposition is considered proven true if we can derive it from the axioms in a finite sequence of logical steps. If the proposition conflicts with another proposition that can be proved to be true, then it is considered false.

Finding such formal systems was a major goal of mathematics at the end of the nineteenth century. Two concerns immediately arose. The first was that the system should be **consistent**. By this we mean that there should not be any proposition that can be proved to be true by one sequence of steps, then shown to be false by another equally valid argument. Consistency is indispensable in mathematics, and anything derived from an inconsistent system would be contrary to all we agree on. A second concern was whether a system is **complete**, by which we mean that any proposition expressible in the system can be proved to be true or false. For some time it was hoped that consistent and complete systems for all of mathematics could be devised thereby opening the door to rigorous but completely mechanical theorem proving. But this hope was dashed by the work of K. Gödel. In his famous **Incompleteness Theorem**, Gödel showed that any interesting consistent system must be incomplete; that is, it must contain some unprovable propositions. Gödel's revolutionary conclusion was published in 1931.

Gödel's work left unanswered the question of whether the unprovable statements could somehow be distinguished from the provable ones, so that there was still some hope that most of mathematics could be made precise with mechanically verifiable proofs. It was this problem that Turing and other mathematicians of the time, particularly A. Church, S. C. Kleene, and E. Post, addressed. In order to study the question, a variety of formal models of computation were established. Prominent among them were the recursive functions of Church and Kleene and Post systems, but there are many other such systems that have been studied. In this chapter we briefly review some of the ideas that arose out of these studies. There is a wealth of material here that we cannot cover. We will give only a very brief presentation, referring the reader to other references for detail. A quite accessible account of recursive functions and Post systems can be found in Denning, Dennis, and Qualitz (1978), while a good discussion of various other rewriting systems is given in Salomaa (1973) and Salomaa (1985).

The models of computation we study here, as well as others that have been proposed, have diverse origins. But it was eventually found that they were all equivalent in their power to carry out computations. The spirit of this observation is generally called **Church's thesis**. This thesis states that all possible models of computation, if they are sufficiently broad, must be equivalent. It also implies that there is an inherent limitation in this and that there are functions that cannot be expressed in any way that gives an explicit method for their computation. The claim is of course very closely related to Turing's thesis, and the combined notion is sometimes called the **Church-Turing thesis**. It provides a general principle for algorithmic computation and, while not provable, gives strong evidence that no more powerful models can be found.

13.1 Recursive Functions

The concept of a function is fundamental to much of mathematics. As summarized in Section 1.1, a function is a rule that assigns to an element of one set, called the **domain** of the function, a unique value in another set, called the **range** of the function. This is very broad and general and immediately raises the question of how we can explicitly represent this association. There are many ways in which functions can be defined. Some of them we use frequently, while others are less common.

We are all familiar with functional notation in which we write expressions like

$$f(n) = n^2 + 1.$$

This defines the function f by means of a recipe for its computation: given any value for the argument n, multiply that value by itself, and then add one. Since the function is defined in this explicit way, we can compute its values in a strictly mechanical fashion. To complete the definition of f, we also must specify its domain. If, for example, we take the domain to be the set of all integers, then the range of f will be some subset of the set of positive integers.

Since many very complicated functions can be specified this way, we may well ask to what extent the notation is universal. If a function is defined (that is, we know the relation between the elements of its domain and its range), can it be expressed in such a functional form? To answer the question, we must first clarify what the permissible forms are: for this we introduce some basic functions, together with rules for building from them some more complicated ones.

Primitive Recursive Functions

To keep the discussion simple, we will consider only functions of one or two variables, whose domain is either I, the set of all non-negative integers, or $I \times I$, and whose range is in I. In this setting, we start with the basic functions:

1. The **zero function** $z(x) = 0$, for all $x \in I$.
2. The **successor function** $s(x)$, whose value is the integer next in sequence to x, that is, in the usual notation, $s(x) = x + 1$.
3. The **projector functions**

$$p_k(x_1, x_2) = x_k, \qquad k = 1, 2.$$

There are two ways of building more complicated functions from these:

1. **Composition**, by which we construct

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

from defined functions g_1, g_2, h.

2. **Primitive recursion**, by which a function can be defined recursively through

$$f(x, 0) = g_1(x),$$
$$f(x, y + 1) = h(g_2(x, y), f(x, y)),$$

from defined functions g_1, g_2, and h.

We illustrate how this works by showing how the basic operations of integer arithmetic can be constructed in this fashion.

Example 13.1 Addition of integers x and y can be implemented with the function $add\,(x, y)$, defined by

$$add\,(x, 0) = x,$$
$$add\,(x, y+1) = add\,(x, y) + 1.$$

To add 2 and 3, we apply these rules successively:

$$\begin{aligned} add\,(3,2) &= add\,(3,1) + 1 \\ &= (add\,(3,0) + 1) + 1 \\ &= (3+1) + 1 \\ &= 4 + 1 = 5. \end{aligned}$$

■

Example 13.2 Using the *add* function defined in Example 13.1, we can now define multiplication by

$$mult\,(x, 0) = 0,$$
$$mult\,(x, y+1) = add\,(x, mult\,(x, y)).$$

Formally, the second step is an application of primitive recursion, in which h is identified with the *add* function, and $g_2\,(x, y)$ is the projector function $p_1\,(x, y)$.

■

Example 13.3 Substraction is not quite so obvious. First, we must define it, taking into account that negative numbers are not permitted in our system. A kind of subtraction is defined from usual subtraction by

$$x \mathbin{\dot{-}} y = x - y \text{ if } x \geq y,$$
$$x \mathbin{\dot{-}} y = 0 \text{ if } x < y.$$

The operator $\dot{-}$ is sometimes called the **monus**; it defines subtraction so that its range is I.

Now we define the predecessor function

$$\begin{aligned} pred(0) &= 0, \\ pred(y+1) &= y, \end{aligned}$$

and from it, the subtracting function

$$\begin{aligned} subtr(x,0) &= x, \\ subtr(x,y+1) &= pred(subtr(x,y)). \end{aligned}$$

To prove that $5-3=2$, we reduce the proposition by applying the definitions a number of times:

$$\begin{aligned} subtr(5,3) &= pred(subtr(5,2)) \\ &= pred(pred(subtr(5,1))) \\ &= pred(pred(pred(subtr(5,0)))) \\ &= pred(pred(pred(5))) \\ &= pred(pred(4)) \\ &= pred(3) \\ &= 2. \end{aligned}$$

■

In much the same way, we can define integer division, but we will leave the demonstration of it as an exercise. If we accept this as given, we see that the basic arithmetic operations are all constructible by the elementary processes described. With the algebraic operations precisely defined, other more complicated ones can now be constructed, and very complex computations built from the simple ones. We call functions that can be constructed in such a manner primitive recursive.

Definition 13.1

A function is called **primitive recursive** if and only if it can be constructed from the basic functions z, s, p_k, by successive composition and primitive recursion.

Note that if g_1, g_2, and h are total functions, then f defined by composition and primitive recursion is also a total function. It follows from this that every primitive recursive function is a total function on I or $I \times I$.

The expressive power of primitive recursive functions is considerable, and most common functions are primitive recursive. However, not all functions are in this class, as the following argument shows.

Theorem 13.1

Let F denote the set of all functions from I to I. Then there is some function in F that is not primitive recursive.

Proof: Every primitive recursive function can be described by a finite string that indicates how it is defined. Such strings can be encoded and arranged in standard order. Therefore, the set of all primitive recursive functions is countable.

Suppose now that the set of all functions is also countable. We can then write all functions in some order, say, $f_1, f_2, ...$. We next construct a function g defined as

$$g(i) = f_i(i) + 1, \qquad i = 1, 2, ...$$

Clearly, g is well defined and is therefore in F, but equally clearly, g differs from every f_i in the diagonal position. This contradiction proves that F cannot be countable.

Combining these two observations proves that there must be some function in F that is not primitive recursive. ■

Actually, this goes even further; not only are there functions that are not primitive recursive, there are in fact computable functions that are not primitive recursive.

Theorem 13.2

Let C be the set of all total computable functions from I to I. Then there is some function in C that is not primitive recursive.

Proof: By the argument of the previous theorem, the set of all primitive recursive functions is countable. Let us denote the functions in this set as $r_1, r_2, ...$ and define a function g by

$$g(i) = r_i(i) + 1.$$

By construction, the function g differs from every r_i and is therefore not primitive recursive. But clearly g is computable, proving the theorem. ■

The nonconstructive proof that there are computable functions that are not primitive recursive is a fairly simple exercise in diagonalization. The actual construction of an example of such a function is a much more complicated matter. We will give here one example that looks quite simple; however, the demonstration that it is not primitive recursive is quite lengthy.

Ackermann's Function

Ackermann's function is a function from $I \times I$ to I, defined by

$$\begin{aligned} A(0,y) &= y+1, \\ A(x,0) &= A(x-1,1), \\ A(x,y+1) &= A(x-1,A(x,y)). \end{aligned}$$

It is not hard to see that A is a total, computable function. In fact, it is quite elementary to write a recursive computer program for its computation. But in spite of its apparent simplicity, Ackermann's function is not primitive recursive.

Of course, we cannot argue directly from the definition of A. Even though this definition is not in the form required for a primitive recursive function, it is possible that an appropriate alternative definition could exist. The situation here is similar to the one we encountered when we tried to prove that a language was not regular or not context-free. We need to appeal to some general property of the class of all primitive recursive functions and show that Ackermann's function violates this property. For primitive recursive functions, one such property is the growth rate. There is a limit to how fast a primitive recursive function can grow as $i \to \infty$, and Ackermann's function violates this limit. That Ackermann's function grows very rapidly is easily demonstrated; see, for example, Exercises 9 to 11 at the end of this section. How this is related to the limit of growth for primitive recursive functions is made precise in the following theorem. Its proof, which is tedious and technical, will be omitted.

Theorem 13.3 Let f be any primitive recursive function. Then there exists some integer n such that

$$f(i) < A(n,i),$$

for all $i = n, n+1, \ldots$.

Proof: For the details of the argument, see Denning, Dennis, and Qualitz (1978, p. 534). ■

If we accept this result, it follows easily that Ackermann's function is not primitive recursive.

Theorem 13.4 Ackermann's function is not primitive recursive.

Proof: Consider the function

$$g(i) = A(i,i).$$

If A were primitive recursive, then so would g. But then, according to Theorem 13.3, there exists an n such that

$$g(i) < A(n,i),$$

for all i. If we now pick $i = n$, we get the contradiction

$$\begin{aligned} g(n) &= A(n,n) \\ &< A(n,n), \end{aligned}$$

proving that A cannot be primitive recursive. ■

μ-Recursive Functions

To extend the idea of recursive functions to cover Ackermann's function and other computable functions, we must add something to the rules by which such functions can be constructed. One way is to introduce the μ or **minimalization** operator, defined by

$$\mu y(g(x,y)) = \text{ smallest } y \text{ such that } g(x,y) = 0.$$

In this definition, we assume that g is a total function.

Example 13.4 Let

$$g(x,y) = x + y - 3,$$

which is a total function. If $x \leq 3$, then

$$y = 3 - x$$

is the result of the minimalization, but if $x > 3$, then there is no $y \in I$ such that $x + y - 3 = 0$. Therefore,

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

We see from this that even though $g(x,y)$ is a total function, $\mu y(g(x,y))$ may only be partial. ■

As the above example shows, the minimalization operation opens the possibility of defining partial functions recursively. But it turns out that it also extends the power to define total functions so as to include all computable functions. Again, we merely quote the major result with references to the literature where the details may be found.

Definition 13.2

A function is said to be μ-recursive if it can be constructed from the basis functions by a sequence of applications of the μ-operator and the operations of composition and primitive recursion.

Theorem 13.5 A function is μ-recursive if and only if it is computable.

Proof: For a proof, see Denning, Dennis, and Qualitz (1978, Chapter 13). ■

The μ-recursive functions therefore give us another model for algorithmic computation.

Exercises

1. Use the definitions in Examples 13.1 and 13.2 to prove that $3 + 4 = 7$ and $2 * 3 = 6$.
2. Define the function

$$\begin{aligned} greater\,(x, y) &= 1 \text{ if } x > y \\ &= 0 \text{ if } x \leq y \end{aligned}$$

Show that this function is primitive recursive. ●

3. Consider the function

$$\begin{aligned} equals\,(x, y) &= 1 \quad \text{if } x = y, \\ &= 0 \quad \text{if } x \neq y. \end{aligned}$$

Show that this function is primitive recursive.

4. Let f be defined by

$$\begin{aligned} f\,(x, y) &= x \quad \text{if } x \neq y, \\ &= 0 \quad \text{if } x = y. \end{aligned}$$

Show that this function is primitive recursive.

★ **5.** Integer division can be defined by two functions *div* and *rem*:

$$div\,(x, y) = n,$$

where n is the largest integer such that $x \geq ny$, and

$$rem\,(x, y) = x - ny.$$

Show that the functions *div* and *rem* are primitive recursive.

6. Show that

$$f\,(n) = 2^n$$

is primitive recursive.

7. Show that the function

$$g\,(x, y) = x^y$$

is primitive recursive.

8. Write a computer program for computing Ackermann's function. Use it to evaluate $A\,(2, 5)$ and $A\,(3, 3)$.

9. Prove the following for the Ackermann function.

(a) $A\,(1, y) = y + 2$

(b) $A\,(2, y) = 2y + 3$

(c) $A\,(3, y) = 2^{y+3} - 3$

10. Use Exercise 9 to compute $A\,(4, 1)$ and $A\,(4, 2)$.

11. Give a general expression for $A\,(4, y)$.

12. Show the sequence of recursive calls in the computation of $A\,(5, 2)$.

13. Show that Ackermann's function is a total function in $I \times I$.

14. Try to use the program constructed for Exercise 8 to evaluate $A\,(5, 5)$. Can you explain what you observe?

15. For each g below, compute $\mu y\,(g\,(x, y))$, and determine its domain.

(a) $g\,(x, y) = xy$

(b) $g\,(x, y) = 2^x + y - 3$

(c) $g\,(x, y) =$ integer part of $(x - 1) / (y + 1)$

(d) $g\,(x, y) = x \bmod (y + 1)$

16. The definition of *pred* in Example 13.3, although intuitively clear, does not strictly adhere to the definition of a primitive recursive function. Show how the definition can be rewritten so that it has the correct form.

13.2 Post Systems

A Post system looks very much like an unrestricted grammar, consisting of an alphabet and some production rules by which successive strings can be derived. But there are significant differences in the way in which the productions are applied.

Definition 13.3

A Post system Π is defined by

$$\Pi = (C, V, A, P),$$

where

C is a finite set of constants, consisting of two disjoint sets C_N, called the **nonterminal constants**, and C_T, the set of **terminal constants**,

V is a finite set of variables,

A is a finite set from C^*, called the **axioms**,

P is a finite set of productions.

The productions in a Post system must satisfy certain restrictions. They must be of the form

$$x_1V_1x_2 \cdots V_nx_{n+1} \rightarrow y_1W_1y_2 \cdots W_my_{m+1}, \tag{13.1}$$

where $x_i,\ y_i \in C^*$, and $V_i,\ W_i \in V$, subject to the requirement that any variable can appear at most once on the left, so that

$$V_i \neq V_j \text{ for } i \neq j,$$

and that each variable on the right must appear on the left, that is

$$\bigcup_{i=1}^{m} W_i \subseteq \bigcup_{i=1}^{n} V_i.$$

Suppose we have a string of terminals of the form $x_1w_1x_2w_2 \cdots w_nx_{n+1}$, where the substrings $x_1,\ x_2 \cdots$ match the corresponding strings in (13.1) and $w_i \in C^*$. We can then make the identification $w_1 = V_1,\ w_2 = V_2, \ldots,$ and substitute these values for the W's on the right of (13.1). Since every

W is some V_i that occurs on the left, it is assigned a unique value, and we get the new string $y_1 w_i y_2 w_j \cdots y_{m+1}$. We write this as

$$x_1 w_1 x_2 w_2 \cdots x_{n+1} \Rightarrow y_1 w_i y_2 w_j \cdots y_{m+1}.$$

As for a grammar, we can now talk about the language derived by a Post system.

Definition 13.4

The language generated by the Post system $\Pi = (C, V, A, P)$ is

$$L(\Pi) = \left\{ w \in C_T^* : w_0 \overset{*}{\Rightarrow} w \text{ for some } w_0 \in A \right\}.$$

Example 13.5 Consider the Post system with

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

and production

$$V_1 \to aV_1b.$$

This allows the derivation

$$\lambda \Rightarrow ab \Rightarrow aabb.$$

In the first step, we apply (13.1) with the identification $x_1 = \lambda$, $V_1 = \lambda$, $x_2 = \lambda$, $y_1 = a$, $W_1 = V_1$, and $y_2 = b$. In the second step, we re-identify $V_1 = ab$, leaving everything else the same. If you continue with this, you will quickly convince yourself that the language generated by this particular Post system is $\{a^n b^n : n \geq 0\}$. ■

Example 13.6 Consider the Post system with

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

and productions

$$V_1 + V_2 = V_3 \to V_1 1 + V_2 = V_3 1,$$
$$V_1 + V_2 = V_3 \to V_1 + V_2 1 = V_3 1.$$

The system allows the derivation

$$\begin{aligned} 1 + 1 = 11 &\Rightarrow 11 + 1 = 111 \\ &\Rightarrow 11 + 11 = 1111. \end{aligned}$$

Interpreting the strings of 1's as unary representations of integers, the derivation can be written as

$$1 + 1 = 2 \Rightarrow 2 + 1 = 3 \Rightarrow 2 + 2 = 4.$$

The language generated by this Post system is the set of all identities of integer additions, such as $2 + 2 = 4$, derived from the axiom $1 + 1 = 2$. ■

Example 13.6 illustrates in a simple manner the original intent of Post systems as a mechanism for rigorously proving mathematical statements from a set of axioms. It also shows the inherent awkwardness of such a completely rigorous approach and why it is rarely used. But Post systems, even though they are cumbersome for proving complicated theorems, are general models for computation, as the next theorem shows.

Theorem 13.6 A language is recursively enumerable if and only if there exists some Post system that generates it.

Proof: The arguments here are relatively simple and we sketch them briefly. First, since a derivation by a Post system is completely mechanical, it can be carried out on a Turing machine. Therefore, any language generated by a Post system is recursively enumerable.

For the converse, remember that any recursively enumerable language is generated by some unrestricted grammar G, having productions all of the form

$$x \to y,$$

with x, $y \in (V \cup T)^*$. Given any unrestricted grammar G, we create a Post system $\Pi = (V_\Pi, C, A, P_\Pi)$, where $V_\Pi = \{V_1, V_2\}, C_N = V, C_T = T, A = \{S\}$, and with productions

$$V_1 x V_2 \to V_1 y V_2,$$

for every production $x \to y$ of the grammar. It is then an easy matter to show that a w can be generated by the Post system Π if and only if it is in the language generated by G. ■

EXERCISES

1. For $\Sigma = \{a, b, c\}$, find a Post system that generates the following languages.

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

   (b) $L = \{ww\}$ ●

   (c) $L = \{a^n b^n c^n\}$

2. Find a Post system that generates

$$L = \left\{ww^R : w \in \{a, b\}^*\right\}.$$

3. For $\Sigma = \{a\}$, what language does the Post system with axiom $\{a\}$ and the following production generate?

$$V_1 \rightarrow V_1 V_1$$ ●

4. What language does the Post system in Exercise 3 generate if the axiom set is $\{a, ab\}$?

5. Find a Post system for proving the identities of integer multiplication, starting from the axiom $1 * 1 = 1$. ●

6. Give the details of the proof of Theorem 13.6.

7. What language does the Post system with

$$V \rightarrow aVV$$

   and axiom set $\{ab\}$ generate?

8. A restricted Post system is one on which every production $x \rightarrow y$ satisfies, in addition to the usual requirements, the further restriction that the number of variable occurrences on the right and left is the same, i.e., $n = m$ in (13.1). Show that for every language L generated by some Post system, there exists a restricted Post system to generates L.

13.3 Rewriting Systems

The various grammars we have studied have a number of things in common with Post systems: They are all based on some alphabet from which one string can be obtained from another. Even a Turing machine can be viewed this way, since its instantaneous description is a string that completely defines its configuration. The program is then just a set of rules for producing one such string from a previous one. These observations can be formalized in the concept of a **rewriting system.** Generally, a rewriting system consists of an alphabet Σ and a set of rules or productions by which a string in Σ^+ can produce another. What distinguishes one rewriting system from another is the nature of Σ and restrictions for the application of the productions.

The idea is quite broad and allows any number of specific cases in addition to the ones we have already encountered. Here we briefly introduce some less well-known ones that are interesting and also provide general models for computation. For details, see Salomaa (1973) and Salomaa (1985).

Matrix Grammars

Matrix grammars differ from the grammars we have previously studied (which are often called **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

$$x_1 \to y_1, x_2 \to y_2, \ldots.$$

Whenever the first production of some set P_i is applied, we must next apply the second one to the string just created, then the third one, and so on. We cannot apply the first production of P_i unless all other productions in this set can also be applied.

Example 13.7 Consider the matrix grammar

$$\begin{aligned} &P_1 : S \to S_1S_2,\\ &P_2 : S_1 \to aS_1, S_2 \to bS_2c,\\ &P_3 : S_1 \to \lambda, S_2 \to \lambda. \end{aligned}$$

A derivation with this grammar is

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

Note that whenever the first rule of P_2 is used to create an a, the second one also has to be used, producing a corresponding b and c. This makes it easy to see that the set of terminal strings generated by this matrix grammar is

$$L = \{a^nb^nc^n : n \geq 0\}.$$

■

Matrix grammars contain phrase-structure grammars as a special case in which each P_i contains exactly one production. Also, since matrix grammars represent algorithmic processes, they are governed by Church's thesis. We conclude from this that matrix grammars and phrase-structure grammars have the same power as models of computation. But, as Example 13.7 shows, sometimes the use of a matrix grammar gives a much simpler solution than we can achieve with an unrestricted phrase-structure grammar.

Markov Algorithms

A Markov algorithm is a rewriting system whose productions

$$x \rightarrow y$$

are considered ordered. In a derivation, the first applicable production must be used. Furthermore, the leftmost occurrence of the substring x must be replaced by y. Some of the productions may be singled out as terminal productions; they will be shown as

$$x \rightarrow .y.$$

A derivation starts with some string $w \in \Sigma$ and continues until either a terminal production is used or until there are no applicable productions.

For language acceptance, a set $T \subseteq \Sigma$ of terminals is identified. Starting with a terminal string, productions are applied until the empty string is produced.

Definition 13.5

Let M be a Markov algorithm with alphabet Σ and terminals T. Then the set

$$L(M) = \left\{ w \in T^* : w \stackrel{*}{\Rightarrow} \lambda \right\}$$

is the language accepted by M.

Example 13.8 Consider the Markov algorithm with $\Sigma = T = \{a, b\}$ and productions

$$ab \rightarrow \lambda,$$
$$ba \rightarrow \lambda.$$

Every step in the derivation annihilates a substring ab or ba, so

$$L(M) = \left\{ w \in \{a, b\}^* : n_a(w) = n_b(w) \right\}.$$

■

Example 13.9 Find a Markov algorithm for

$$L = \{a^n b^n : n \geq 0\}.$$

An answer is

$$\begin{aligned} ab &\to S, \\ aSb &\to S, \\ S &\to \cdot\lambda. \end{aligned}$$

If in this last example we take the first two productions and reverse the left and right sides, we get a context-free grammar that generates the language L. In a certain sense, Markov algorithms are simply phrase-structure grammars working backward. This cannot be taken too literally, since it is not clear what to do with the last production. But the observation does provide a starting point for a proof of the following theorem that characterizes the power of Markov algorithms. ■

Theorem 13.7 A language is recursively enumerable if and only if there exists a Markov algorithm for it.

Proof: See Salomaa (1985, p. 35). ■

L-Systems

The origins of L-systems are quite different from what we might expect. Their developer, A. Lindenmayer, used them to model the growth pattern of certain organisms. L-systems are essentially *parallel* rewriting systems. By this we mean that in each step of a derivation, every symbol has to be rewritten. For this to make sense, the productions of an L-system must be of the form

$$a \to u, \tag{13.2}$$

where $a \in \Sigma$ and $u \in \Sigma^*$. When a string is rewritten, one such production must be applied to every symbol of the string before the new string is generated.

Example 13.10 Let $\Sigma = \{a\}$ and

$$a \rightarrow aa$$

define an L-system. Starting from the string a, we can make the derivation

$$a \Rightarrow aa \Rightarrow aaaa \Rightarrow aaaaaaaa.$$

The set of strings so derived is clearly

$$L = \left\{a^{2n} : n \geq 0\right\}.$$

Note again how such special rewriting systems are able to deal with problems that are quite difficult for phrase structure grammars.

■

It is known that L-systems with productions of the form (13.2) are not sufficiently general to provide for all algorithmic computations. An extension of the idea provides the necessary generalization. In an extended L-system, productions are of the form

$$(x, a, y) \rightarrow u,$$

where $a \in \Sigma$ and $x, y, u \in \Sigma^*$, with the interpretation that a can be replaced by u only if it occurs as part of the string xay. It is known that such extended L-systems are general models of computation. For details, see Salomaa (1985).

EXERCISES

1. Find a matrix grammar for

$$L = \{ww : w \in \{a, b\}^*\}. ●$$

2. What language is generated by the matrix grammar

$$P_1 : S \rightarrow S_1S_2,$$
$$P_2 : S_1 \rightarrow aS_1b, S_2 \rightarrow bS_2a,$$
$$P_3 : S_1 \rightarrow \lambda, S_2 \rightarrow \lambda.$$

3. Suppose that in Example 13.7 we change the last group of productions to

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

What language is generated by this matrix grammar?

4. Why does the Markov algorithm in Example 13.9 not accept $abab$?

5. Find a Markov algorithm that derives the language $L = \{a^n b^n c^n : n \geq 1\}$. ●

★ 6. Find a Markov algorithm that accepts

$$L = \{a^n b^m a^{nm} : n \geq 1, m \geq 1\}.$$

7. Find an L-system that generates $L(aa^*)$.

8. What is the set of strings generated by the L-system with productions

$$a \rightarrow aa,$$
$$a \rightarrow aaa,$$

when started with the string a? ●