Formal Grammars and Languages

Tao Jiang
Department of Computer Science
McMaster University
Hamilton, Ontario L8S 4K1, Canada

Ming Li
Department of Computer Science
University of Waterloo
Waterloo, Ontario N2L 3G1, Canada

Bala Ravikumar
Department of Computer Science
University of Rhode Island
Kingston, RI 02881, USA

1 Introduction

Formal language theory as a discipline is generally regarded as growing from the work of linguist Noam Chomsky in the 1950s, when he attempted to give a precise characterization of the structure of natural languages. His goal was to define the syntax of languages using simple and precise mathematical rules. Later it was found that the syntax of programming languages can be described using one of Chomsky's grammatical models called context-free grammars. Much earlier, the Norwegian mathematician Axel Thue studied sequences of binary symbols subject to interesting mathematical properties, such as not having the same substring three times in a row. His work influenced Emil Post, Stephen Kleene, and others to study the mathematical properties of strings and collections of strings.

Soon after the advent of modern electronic computers, people realized that all forms of information—whether numbers, names, pictures, or sound waves—can be represented as strings. Then collections of strings known as *languages* became central to computer science. This section is concerned with fundamental mathematical properties of languages and language generating systems, such as grammars. Every

programming language from Fortran to Java can be precisely described by a grammar. Moreover, the grammar allows us to write a computer program (called the *syntax analyzer* in a compiler) to determine whether a string of statements is syntactically correct in the programming language. Many people would wish that natural languages such as English could be analyzed as precisely, that we could write computer programs to tell which English sentences are grammatically correct. Despite recent advances in *natural language processing*, many of which have been spurred by formal grammars and other theoretical tools, today's commercial products for grammar and style fall well short of that ideal. The main problem is that *there is no common agreement* on what are grammatically correct (English) sentences; nor has anyone yet been able to offer a grammar precise enough to propose as definitive. And style is a matter of taste! such as not beginning sentences with "and" or using interior exclamations. Formal languages and grammars have many applications in other fields, including molecular biology (see [Searls, 1993]) and symbolic dynamics (see [Lind and Marcus, 1995]).

In this chapter, we will present some formal systems that define families of formal languages arising in many computer science applications. Our primary focus will be on context-free languages, since they are most widely used to describe the syntax of programming languages. In the rest of this section, we present some basic definitions and terminology.

Definition 1.1 An **alphabet** is a finite nonempty set of *symbols*. Symbols are assumed to be indivisible.

For example, an alphabet for English can consist of as few as the 26 lower-case letters $a, b, \ldots, z$, adding some punctuation symbols if sentences rather than single words will be considered. Or it may include all of the symbols on a standard North American typewriter, which together with terminal control codes yields the 128-symbol ASCII alphabet, in which much of the world's communication takes place. The new world standard is an alphabet called *UNICODE*, which is intended to provide symbols for all the world's languages—as of this writing, over 38,000 symbols have been assigned. But most important aspects of formal languages can be modeled using the simple two-letter alphabet $\{0, 1\}$, over which ASCII and UNICODE are encoded to begin with. We usually use the symbol Σ to denote an alphabet.

Definition 1.2 A **string** over an alphabet Σ is a finite sequence of symbols of Σ.

The number of symbols in a string x is called its *length*, denoted by $|x|$. It is convenient to introduce a notation ϵ for the empty string, which contains no symbols at all. The length of ϵ is 0.

Definition 1.3 Let $x = a_1a_2\cdots a_n$ and $y = b_1b_2\cdots b_m$ be two strings. The *concatenation* of x and y, denoted by xy, is the string $a_1a_2\cdots a_nb_1b_2\cdots b_m$.

Then for any string x, $\epsilon x = x\epsilon = x$. For any string x and integer $n \geq 0$, we use x^n to denote the string formed by sequentially concatenating n copies of x.

Definition 1.4 The set of all strings over an alphabet Σ is denoted by Σ^*, and the set of all nonempty strings over Σ is denoted by Σ^+. The empty set of strings is denoted by $\emptyset$.

Definition 1.5 For any alphabet Σ, a **language** over Σ is a set of strings over Σ. The members of a language are also called the *words* of the language.

Example 1.1 The sets $L_1 = \{01, 11, 0110\}$ and $L_2 = \{0^n1^n | n \geq 0\}$ are two languages over the binary alphabet $\{0,1\}$. L_1 has three words, while L_2 is infinite. The string 01 is in both languages while 11 is in L_1 but not in L_2.

Since languages are just sets, standard set operations such as union, intersection, and complementation apply to languages. It is useful to introduce two more operations for languages: *concatenation* and *Kleene closure*.

Definition 1.6 Let L_1 and L_2 be two languages over Σ. The concatenation of L_1 and L_2, denoted by L_1L_2, is the language $\{xy | x \in L_1, y \in L_2\}$.

Definition 1.7 Let L be a language over Σ. Define $L^0 = \{\epsilon\}$ and $L^i = LL^{i-1}$ for $i \geq 1$. The *Kleene closure* of L, denoted by L^*, is the language

$$L^* = \bigcup_{i \geq 0} L^i.$$

The *positive closure* of L, denoted by L^+, is the language

$$L^+ = \bigcup_{i \geq 1} L^i.$$

In other words, the Kleene closure of a language L consists of all strings that can be formed by concatenating zero or more words from L. For example, if $L = \{0, 01\}$, then $LL = \{00, 001, 010, 0101\}$, and L^* comprises all binary strings in which every 1 is preceded by a 0. Note that concatenating zero words always gives the empty string, and that a string with no 1s in it still makes the condition on "every 1" true. L^+ has the meaning "concatenate *one* or more words from L," and satisfies the properties $L^* = L^+ \cup \{\epsilon\}$ and $L^+ = LL^*$. Furthermore, for any language L, L^* always contains ϵ, and L^+ contains ϵ if and only if L does. Also note that Σ^* is in fact the Kleene closure of the alphabet Σ when Σ is viewed as a language of words of length 1, and Σ^+ is just the positive closure of Σ.

2 Representation of Languages

In general a language over an alphabet Σ is a subset of Σ^*. How can we describe a language rigorously so that we know whether a given string belongs to the language or not? As shown in Eaxmple 1.1, a finite language such as L_1 can be explicitly defined by enumerating its elements. An infinite language such as L_2 cannot be exhaustively enumerated, but in the case of L_2 we were able to give a simple rule characterizing all of its members. In English, the rule is, "some number of 0s followed by an equal number of 1s." Can we find systematic methods for defining rules that characterize a wide class of languages? In the following we will introduce three such methods: **regular expressions**, **pattern systems**, and **grammars**. Interestingly, only the last is capable of specifying the simple rule for L_2, although the first two work for many intricate languages. The term **formal languages** refers to languages that can be described by a body of systematic rules.

2.1 Regular Expressions and Languages

Let Σ be an alphabet.

Definition 2.1 The **regular expressions** over Σ and the languages they represent are defined inductively as follows.

1. The symbol $\emptyset$ is a regular expression, and represents the empty language.
2. The symbol ϵ is a regular expression, and represents the language whose only member is the empty string, namely $\{\epsilon\}$.
3. For each $c \in \Sigma$, c is a regular expression, and represents the language $\{c\}$, whose only member is the string consisting of the single character c.
4. If r and s are regular expressions representing the languages R and S, then $(r+s)$, (rs) and (r^*) are regular expressions that represent the languages $R \cup S$, RS, and R^*, respectively.

For example, $((0(0+1)^*) + ((0+1)^*0))$ is a regular expression over $\{0,1\}$ that represents the language consisting of all binary strings that begin or end with a 0. Since the set operations union and concatenation are both associative, and since we can stipulate that Kleene closure takes precedence over concatenation and concatenation over union, many parentheses can be omitted from regular expressions. For example, the above regular expression can be written as $0(0+1)^* + (0+1)^*0$. We will also abbreviate the expression rr^* as r^+. Let us look at a few more examples of regular expressions and the languages they represent.

Example 2.1 The expression $0(0+1)^*1$ represents the set of all strings that begin with a 0 and end with a 1.

Example 2.2 The expression $0+1+0(0+1)^*0+1(0+1)^*1$ represents the set of all nonempty binary strings that begin and end with the same bit. Note the inclusion of the strings 0 and 1 as special cases.

Example 2.3 The expressions 0^*, 0^*10^*, and $0^*10^*10^*$ represent the languages consisting of strings that contain no 1, exactly one 1, and exactly two 1's, respectively.

Example 2.4 The expressions $(0+1)^*1(0+1)^*1(0+1)^*$, $(0+1)^*10^*1(0+1)^*$, $0^*10^*1(0+1)^*$, and $(0+1)^*10^*10^*$ all represent the same set of strings that contain at least two 1's.

Two or more regular expressions that represent the same language, as in Example 2.4, are called *equivalent*. It is possible to introduce algebraic identities for regular expressions in order to construct equivalent expressions. Two such identities are $r(s+t) = rs + rt$, which says that concatenation distributes over union the same way "times" distributes over "plus" in ordinary algebra (but taking care that concatenation isn't commutative), and $r^* = (r^*)^*$. These two identities are easy to prove; the reader seeking more detail may consult [Salomaa, 1966].

Example 2.5 Let us construct a regular expression for the set of all strings that contain no consecutive 0s. A string in this set may begin and end with a sequence of 1s. Since there are no consecutive 0s, every 0 that is not the last symbol of the string must be followed by a 1. This gives us the expression $1^*(01^+)^*1^*(\epsilon+0)$. It is not hard to see that the second 1^* is redundant and thus the expression can in fact be simplified to $1^*(01^+)^*(\epsilon+0)$.

Regular expressions were first introduced by [Kleene, 1956] for studying the properties of neural nets. The above examples illustrate that regular expressions often give very clear and concise representations of languages. The languages represented by regular expressions are called the *regular languages*. Fortunately or unfortunately, not every language is regular. For example, there are no regular expressions that represent the languages $\{0^n1^n | n \geq 1\}$ or $\{xx \mid x \in \{0,1\}^*\}$; the latter case is proved at the end of Section 2.1 in Chapter 30.

2.2 Pattern Languages

Another way of representing languages is to use *pattern systems* [Angluin, 1980] (see also [Jiang et al., 1995]).

Definition 2.2 A **pattern system** is a triple (Σ, V, p), where Σ is the alphabet, V is the set of *variables* with $\Sigma \cap V = \emptyset$, and p is a string over $\Sigma \cup V$ called the *pattern*.

Definition 2.3 The language generated by a pattern system (Σ, V, p) consists of all strings over Σ that can be obtained from p by replacing each variable in p with a string over Σ.

An example pattern system is $(\{0,1\}, \{v_1, v_2\}, v_1v_10v_2)$. The language it generates contains all words that begin with a 0 (since v_1 can be chosen as the empty string, and v_2 as an arbitrary string), and contains some words that begin with a 1, such as 110 (by taking $v_1 = 1$, $v_2 = \epsilon$) and 101001 (by taking $v_1 = 10$, $v_2 = 1$). However, it does not contain the strings $\epsilon, 1, 10, 11, 100, 101$, etc. The pattern system $(\{0,1\}, \{v_1\}, v_1v_1)$ generates the set of all strings that are the concatenation of two equal substrings, namely the set $\{xx | x \in \{0,1\}^*\}$. The languages generated by pattern systems are called *pattern languages*.

Regular languages and pattern languages are really different. We have noted that the pattern language $\{xx | x \in \{0,1\}^*\}$ is not a regular language, and one can prove that the set represented by the regular expression 0^*1^* is not a pattern language. Although it is easy to write an algorithm to decide whether a given string is in the language generated by a given pattern system, such an algorithm would most likely have to be very inefficient [Angluin, 1980].

2.3 General Grammars

Perhaps the most useful and general system for representing languages is based on the formal notion of a *grammar*.

Definition 2.4 A **grammar** is a quadruple (Σ, V, S, P), where:

1. Σ is a finite nonempty set called the **terminal alphabet**. The elements of Σ are called the **terminals**.

2. V is a finite nonempty set disjoint from Σ. The elements of V are called the **nonterminals** or **variables**.

3. $S \in V$ is a distinguished nonterminal called the **start symbol**.

4. P is a finite set of **productions** (or **rules**) of the form

$$\alpha \to \beta$$

where $\alpha \in (\Sigma \cup V)^*V(\Sigma \cup V)^*$ and $\beta \in (\Sigma \cup V)^*$, *i.e.* α is a string of terminals and nonterminals containing at least one nonterminal and β is a string of terminals and nonterminals.

Example 2.6 Let $G_1 = (\{0,1\}, \{S,T,O,I\}, S, P)$, where P contains the following productions

$$\begin{aligned} S &\to OT \\ S &\to OI \\ T &\to SI \\ O &\to 0 \\ I &\to 1 \end{aligned}$$

As we shall see, the grammar G_1 can be used to describe the set $\{0^n1^n | n \geq 1\}$.

Example 2.7 Let $G_2 = (\{0,1,2\}, \{S,A\}, S, P)$, where P contains the following productions

$$\begin{aligned} S &\to 0SA2 \\ S &\to \epsilon \\ 2A &\to A2 \\ 0A &\to 01 \\ 1A &\to 11 \end{aligned}$$

This grammar G_2 can be used to describe the set $\{0^n1^n2^n \geq n \geq 0\}$.

Example 2.8 To construct a grammar G_3 to describe English sentences, one might let the alphabet Σ comprise all English *words* rather than letters. V would contain nonterminals that correspond to the structural components in an English sentence, such as <sentence>, <subject>, <predicate>, <noun>, <verb>, <article>, and so on. The start symbol would be <sentence>. Some typical productions are:

<sentence> $\to$ <subject><predicate>
<subject> $\to$ <noun>
<predicate> $\to$ <verb><article><noun>
<noun> $\to$ mary
<noun> $\to$ algorithm
<verb> $\to$ wrote
<article> $\to$ an

The rule <sentence> $\to$ <subject><predicate> models the fact that a sentence can consist of a subject phrase and a predicate phrase. The rules <noun> $\to$ mary and <noun> $\to$ algorithm mean that both "mary" and "algorithm" are possible nouns. This approach to grammar, stemming from Chomsky's work, has influenced even elementary-school teaching.

To explain how a grammar represents a language, we need the following concepts.

Definition 2.5 Let (Σ, V, S, P) be a grammar. A **sentential form** of G is any string of terminals and nonterminals, i.e. a string over $\Sigma \cup V$.

Definition 2.6 Let (Σ, V, S, P) be a grammar, and let γ_1, γ_2 be two sentential forms of G. We say that γ_1 **directly derives** γ_2, written $\gamma_1 \Rightarrow \gamma_2$, if $\gamma_1 = \sigma\alpha\tau$, $\gamma_2 = \sigma\beta\tau$, and $\alpha \to \beta$ is a production in P.

For example, the sentential form $00S11$ directly derives the sentential form $00OT11$ in grammar G_1, and $A2A2$ directly derives $AA22$ in grammar G_2.

Definition 2.7 Let γ_1 and γ_2 be two sentential forms of a grammar G. We say that γ_1 **derives** γ_2, written $\gamma_1 \Rightarrow^* \gamma_2$, if there exists a sequence of (zero or more) sentential forms $\sigma_1, \ldots, \sigma_n$ such that

$$\gamma_1 \Rightarrow \sigma_1 \Rightarrow \cdots \Rightarrow \sigma_n \Rightarrow \gamma_2.$$

The sequence $\gamma_1 \Rightarrow \sigma_1 \Rightarrow \cdots \Rightarrow \sigma_n \Rightarrow \gamma_2$ is called a **derivation** of γ_2 from γ_1.

For example, in grammar G_1, $S \Rightarrow^* 0011$ because

$$S \Rightarrow \underline{O}T \Rightarrow 0\underline{T} \Rightarrow 0S\underline{I} \Rightarrow 0\underline{S}1 \Rightarrow 0\underline{O}I1 \Rightarrow 00\underline{I}1 \Rightarrow 0011$$

and in grammar G_2, $S \Rightarrow^* 001122$ because

$$S \Rightarrow 0\underline{S}A2 \Rightarrow 00\underline{S}A2A2 \Rightarrow 0\underline{0A}2A2 \Rightarrow 001\underline{2A}2 \Rightarrow 001\underline{1A}22 \Rightarrow 001122.$$

Here the left-hand side of the relevant production in each derivation step is underlined for clarity.

Definition 2.8 Let (Σ, V, S, P) be a grammar. The language generated by G, denoted by $L(G)$, is defined as

$$L(G) = \{x | x \in \Sigma^*, S \Rightarrow^* x\}.$$

The words in $L(G)$ are also called the *sentences* of $L(G)$.

Clearly, $L(G_1)$ contains all strings of the form 0^n1^n, $n \geq 1$, and $L(G_2)$ contains all strings of the form $0^n1^n2^n$, $n \geq 0$. Although only a partial definition of G_3 is given, we know that $L(G_3)$ contains sentences like "mary wrote an algorithm" and "algorithm wrote an algorithm," but does not contain strings like "an wrote algorithm."

Formal grammars were introduced as such by [Post, 1943], and had antecedents in work by Thue and others. However, the study of their rigorous use in describing formal (and natural) languages did not begin until the mid-1950s [Chomsky, 1956]. In the next section, we consider various restrictions on the form of productions in a grammar, and see how these restrictions can affect its power to represent languages. In particular, we will show that regular languages and pattern languages can all be generated by grammars under different restrictions.

3 Hierarchy of Grammars

Grammars can be divided into four classes by gradually increasing the restrictions on the form of the productions. Such a classification is due to Chomsky [Chomsky, 1956, Chomsky, 1963] and is called the *Chomsky hierarchy*.

DEFINITION 3.1 Let $G = (\Sigma, V, S, P)$ be a grammar.

1. G is also called a **Type-**0 grammar or an **unrestricted** grammar.

2. G is a **Type-1** or **context-sensitive** grammar if each production $\alpha \to \beta$ in P satisfies $|\alpha| \le |\beta|$. By "special dispensation," we also allow a Type-1 grammar to have the production $S \to \epsilon$, provided S does not appear on the right-hand side of any production.

3. G is a **Type-2** or **context-free** grammar if each production $\alpha \to \beta$ in P satisfies $|\alpha| = 1$; i.e., α is a single nonterminal.

4. G is a **Type-**3 or **right-linear** or **regular** grammar if each production has one of the following three forms:

$$A \to cB, \quad A \to c, \quad A \to \epsilon,$$

   where A, B are nonterminals (with $B = A$ allowed) and c is a terminal.

The language generated by a Type-i grammar is called a Type-i language, $i = 0, 1, 2, 3$. A Type-1 language is also called a **context-sensitive language** (CSL), and a Type-2 language is also called a **context-free language** (CFL). The "special dispensation" allows a CSL to contain ϵ, and thus allows one to say that every CFL is also a CSL. Many sources allow "right-linear" grammars to have productions of the form $A \to xB$, where x is any string of terminals, and/or exclude one of the forms $A \to c$, $A \to \epsilon$ from their definition of "regular" grammar (perhaps allowing $S \to \epsilon$ in the latter case). Regardless of the choice of definitions, every Type-3 grammar generates a regular language, and every regular language has a Type-3 grammar; we have proved this using finite automata in Chapter 30. Stated in other words:

THEOREM 3.1 *The class of Type-*3 *languages and the class of regular languages are equal.*

The grammars G_1 and G_3 given in the last section are context-free and the grammar G_2 is context-sensitive. Now we give some examples of unrestricted and right-linear grammars.

EXAMPLE 3.1 Let $G_4 = (\{0,1\}, \{S, A, O, I, T\}, S, P)$, where P contains

$$\begin{array}{rcl@{\qquad}rcl}
S & \to & AT & & & \\
A & \to & 0AO & A & \to & 1AI \\
O0 & \to & 0O & O1 & \to & 1O \\
I0 & \to & 0I & I1 & \to & 1I \\
OT & \to & 0T & IT & \to & 1T \\
A & \to & \epsilon & T & \to & \epsilon
\end{array}$$

Then G_4 generates the set $\{xx | x \in \{0,1\}^*\}$. To understand how this grammar works, think of the nonterminal O as saying, "I must ensure that the right half gets a terminal 0 in the same place as the terminal 0 in the production $A \to 0AO$ that introduced me." The nonterminal I eventually forces the precise placement of a terminal 1 in the right-hand side in the same way. The nonterminal T makes sure that O and I place their 0 and 1 on the right-hand side rather than prematurely. Only after every O and I has moved right past any earlier-formed terminals 0 and 1 and been eliminated "in the context of" T, and the production $A \to \epsilon$ is used to signal that no additional O or I will be introduced, can the endmarker T be dispensed with via $T \to \epsilon$. For example, we can derive the word 0101 from S as follows:

$$S \Rightarrow \underline{A}T \Rightarrow 0\underline{A}OT \Rightarrow 01\underline{A}IOT \Rightarrow 01I\underline{OT} \Rightarrow 01\underline{I0}T \Rightarrow 010\underline{IT} \Rightarrow 0101\underline{T} \Rightarrow 0101.$$

Only the productions $A \to \epsilon$ and $T \to \epsilon$ prevent this grammar from being Type-1. The interested reader is challenged to write a Type-1 grammar for this language.

EXAMPLE 3.2 We give a right-linear grammar G_5 to generate the language represented by the regular expression in Example 2.2, i.e., the set of all nonempty binary strings beginning and ending with the same bit. Let $G_5 = (\{0,1\}, \{S, O, I\}, S, P)$, where P contains

$$\begin{array}{rcl@{\qquad}rcl}
S & \to & 0O & S & \to & 1I \\
S & \to & 0 & S & \to & 1 \\
O & \to & 0O & O & \to & 1O \\
I & \to & 0I & I & \to & 1I \\
O & \to & 0 & I & \to & 1
\end{array}$$

Here O means to remember that the last bit must be a 0, and 1 similarly forces the last bit to be a 1. Note again how the grammar treats the words 0 and 1 as special cases.

Every regular grammar is a context-free grammar, but not every context-free grammar is context-sensitive. However, every context-free grammar G can be transformed into an equivalent one in which every production has the form $A \to BC$ or $A \to c$, where A, B, and C are (possibly identical) variables, and c is a terminal. If the empty string is in $L(G)$, then we can arrange to include $S \to \epsilon$ under the same "special dispensation" as for CSLs. This form is called **Chomsky normal form** [Chomsky, 1963], where it was used to prove the case $i = 1$ of the next theorem. The grammar G_1 in the last section is an example of a context-free grammar in Chomsky normal form.

THEOREM 3.2 *For each $i = 0, 1, 2$, the class of Type-i languages properly contains the class of Type-$(i + 1)$ languages.*

The containments are clear from the above remarks. For the proper containments, we have already seen that $\{0^n 1^n | n \geq 0\}$ is a Type-2 language that is not regular, and Chapter 32 will show that the language of the Halting Problem is Type-0 but not Type-1. One can prove by a technique called "pumping" that the Type-1 languages $\{0^n 1^n 2^n | n \geq 0\}$ and $\{xx | x \in \{0, 1\}^*\}$ are not Type-2. See [Hopcroft and Ullman, 1979] for this, and for a presentation of the algorithm for converting a context-free grammar into Chomsky normal form.

The four classes of languages in the Chomsky hierarchy have also been completely characterized in terms of Turing machines (see Chapter 30) and natural restrictions on them. We mention this here to make the point that these characterizations show that these classes capture fundamental properties of computation, not just of formal languages. A *linear bounded automaton* is a possibly-nondeterministic Turing machine that on any input x uses only the cells initially occupied by x, except for one visit to the blank cell immediately to the right of x (which is the initially-scanned cell if $x = \epsilon$). Pushdown automata may also be nondeterministic and were likewise introduced in Chapter 30.

THEOREM 3.3

(a) The class of Type-0 languages equals the class of languages accepted by Turing machines.

(b) The class of Type-1 languages equals the class of languages accepted by linear bounded automata.

(c) The class of Type-2 languages equals the class of languages accepted by pushdown automata.

(d) The class of Type-3 languages equals the class of languages accepted by finite automata.

Proof. (a) Given a Type-0 grammar G, one can build a nondeterministic Turing machine M that accepts $L(G)$ by having M first write the start symbol S of G on a second tape. M always nondeterministically chooses a production and chooses a place (if any) on its second tape where it can be applied. If and when the second tape becomes an all-terminal string, M compares it to its input, and if they match, M accepts. Then $L(M) = L(G)$, and by Theorem 2.4 of Chapter 30, M can be converted into an equivalent deterministic single-tape Turing machine.

For the reverse simulation of a TM by a grammar we give full details. Given any TM M_0, we may modify M_0 into an equivalent TM $M = (Q, \Sigma, \Gamma, \delta, B, q_0, q_f)$ that has the following five properties: (i) M never writes a blank; (ii) M when reading a blank always converts it to a non-blank symbol on the current step; (iii) M begins with a transition from q_0 that overwrites the first input cell (remembering what it was) by a special symbol $\wedge$ that is never altered; (iv) M never re-enters state q_0 or moves left of $\wedge$; and (v) whenever M is about to accept, M moves left to the $\wedge$, where it executes an instruction that moves right and enters a distinguished state q_e. In state q_e it overwrites any non-blank character by a special new symbol # and moves right; when it hits the blank after having #-ed out the rightmost non-blank symbol on its tape, M finally goes to q_f and accepts.

Given M with these properties, take $V = \{S, A, \} \cup (Q \times \Gamma) \cup (\Gamma \setminus \Sigma)$. A single symbol in $Q \times \Gamma$ is written using square brackets; e.g. $[q, c]$ means that M is in state q scanning character c. The grammar G has the following productions, which intuitively can simulate any accepting computation by M in reverse:

(1) $S \to \wedge S_0; \quad S_0 \to \# S_0 \mid [q_e, \#]$;

(2) $[r, d] \to [q, c]$, for all instructions $(q, c, d, r) \in \delta$ with $q, r \in Q$ and $c, d \in \Gamma$;

(3) $c[r, B] \to [q, c]A$, for all $(q, c, R, r) \in \delta$;

(4) $c[r, d] \to [q, c]d$, for all $(q, c, R, r) \in \delta$ and $d \in \Gamma$, $d \neq B$;

(5) $[r, d]c \to d[q, c]$, for all $(q, c, L, r) \in \delta$ and $d \in \Gamma$, $d \neq B$;

(6) $[q_0, c] \to c$ for all $c \in \Sigma$, and

(7) $A \to \epsilon$.

For all $x \in L(M)$, G can generate x by first using the productions in (1) to lay down a # for every cell *used* during the computation, using the productions (2)–(5) to simulate the computation in reverse, using (6) to restore the first bit of x (blank if $x = \epsilon$) one step after having eliminated the nonterminal $\wedge$, and using (7) to erase each A marking an initially-blank cell that M used. Conversely, the only way G can eliminate $\wedge$ and reach an all-terminal string is by winding back an accepting computation of M all the way to state q_0 scanning the first cell. Hence $L(G) = L(M)$.

(b) If the given TM M_0 is a linear bounded automaton, then we can patch the last construction to eliminate the productions in (3) and (7), yielding a context-sensitive grammar G. To do this, we need to make M postpone its one allowed visit to the blank cell after the input until the last step of an accepting computation. To do this, we make M nondeterministically guess which bit of its input x is the last one, and overwrite it by an immutable right endmarker \$ the same way it did with $\wedge$ on the left. Then we arrange that from state q_e, M will accept only if it sees a blank immediately to the right of the \$, meaning that its initial guess delimited exactly the true input x. (Technically this needs another state q_e'.) Now M never even scans a blank in the middle of an accepting computation, and we can delete the productions in (3) as well as (7). Moreover, if M_0 accepts ϵ, we can add the production $S \to \epsilon$ allowed by the "special dispensation" for context-sensitive grammars above.

Going the other way, if the grammar G in the first paragraph of this proof is context-sensitive, then the resulting TM M uses only $O(n)$ space, and can be converted to an equivalent linear bounded automaton by Theorem 3.1 of Chapter 30.

(c) Given a context-free grammar G, we may assume that G is in Chomsky normal form. We can build a nondeterministic PDA M whose initial moves lay down a bottom-of-stack marker $\wedge$ and the start symbol S of G, and go to a "central" state q. For every production of the form $A \to BC$ in G, M has moves that pop the stack if A is uppermost and push C and then B, returning to state q. For every production of the form $A \to c$, M can pop an uppermost A from its stack if the currently-scanned input symbol is c; then it moves its input head right. If G has the production $S \to \epsilon$ as a special case, then M can pop the initial S. A computation path accepts if and only if the stack gets down to $\wedge$ precisely when M reaches the blank at the end of its input x. Then accepting paths of M on an input x are in 1-1 correspondence with leftmost derivations (see below) of x in G, so $L(M) = L(G)$.

Going from a PDA M to an equivalent CFG G is much trickier, and is covered well in [Hopcroft and Ullman, 1979].

(d) This has been proved in Chapter 30, Theorem 2.2.

Since $\{xx | x \in \{0,1\}^*\}$ is a pattern language, we know from discussions above that the class of pattern languages is not contained in the class of context-free languages. It is contained in the class of context-sensitive languages, however.

THEOREM 3.4 *Every pattern language is context-sensitive.*

This was proved by showing that every pattern language is accepted by a linear bounded automaton [Angluin, 1980], whereupon it is a corollary of Theorem 3.3(b).

Given a class of languages, we are often interested in the so called *closure properties* of the class.

DEFINITION 3.2 A class of languages is said to be *closed under* a particular operation (such as union, intersection, complementation, concatenation, or Kleene closure) if eevery application of the operation on language(s) of the class yields a language of the class.

Closure properties are often useful in constructing new languages from existing languages, and for proving many theoretical properties of languages and grammars. The closure properties of the four types of languages in the Chomsky hierarchy are summarized below. Proofs may be found in [Harrison, 1978], [Hopcroft and Ullman, 1979], or [Gurari, 1989]; the closure of the CSLs under complementation is the famous Immerman-Szelepcsényi Theorem, which is treated in Chapter 33, Section 2.5.

THEOREM 3.5

1. *The class of Type-*0 *languages is closed under union, intersection, concatenation, and Kleene closure, but not under complementation.*

2. *The class of context-free languages is closed under union, concatenation and Kleene closure, but not under intersection or complementation.*

3. *The classes of context-sensitive and regular languages are closed under all of the five operations.*

For example, let $L_1 = \{0^m 1^n 2^p | m = n\}$, $L_2 = \{0^m 1^n 2^p | n = p\}$, and $L_3 = \{0^m 1^n 2^p | m = n \text{ or } n = p\}$. Now L_1 is the concatenation of the context-free languages $\{0^n 1^n | n \geq 0\}$ and 2^*, so L_1 is context-free. Similarly L_2 is context-free. Since $L_3 = L_1 \cup L_2$, L_3 is context-free. However, intersecting L_1 with L_2 gives the language $\{0^m 1^n 2^p | m = n = p\}$, which is not context-free.

We will look at context-free grammars more closely in the next section and introduce the concepts of parsing and ambiguity.

4 Context-free Grammars and Parsing

From a practical point of view, for each grammar $G = (\Sigma, V, S, P)$ representing some language, the following two problems are important:

1. **Membership problem**: Given a string over Σ, does it belong to $L(G)$?

2. **Parsing problem**: Given a string in $L(G)$, how can it be derived from S?

The importance of the membership problem is quite obvious—given an English sentence or computer program, we wish to know if it is grammatically correct or has the right format. Solving the membership problem for context-free grammars is an integral step in the *lexical analysis* of computer programs, namely the stage of decomposing each statement into *tokens*, prior to fully parsing the program. For this reason, the membership problem is also often referred to as lexical analysis (cf. [Drobot, 1989]). Parsing is important because a derivation usually brings out the "meaning" of the string. For example, in the case of a Pascal program, a derivation of the program in the Pascal grammar tells the compiler how the program should be executed. The following theorem qualifies the decidability of the membership problem for the four classes of grammars in the Chomsky hierarchy. Proofs of the first assertion can be found in [Chomsky, 1963, Harrison, 1978, Hopcroft and Ullman, 1979], while the second assertion is treated below. Decidability and time complexity were defined in Chapter 30.

Theorem 4.1 *The membership problem for Type-0 grammars is undecidable in general, but it is decidable given any context-sensitive grammar. For context-free grammars the problem is decidable in polynomial time, and for regular grammars, linear time.*

Since context-free grammars play a very important role in describing computer programming languages, we discuss the membership and parsing problems for context-free grammars in more detail in this and the next section. First, let us look at another example of a context-free grammar. For convenience, let us abbreviate a set of productions

$$A \to \alpha_1, \ldots, A \to \alpha_n$$

with the same left-hand side nonterminal as

$$A \to \alpha_1 | \ldots | \alpha_n.$$

Example 4.1 We construct a context-free grammar G_6 for the set of all valid real-number literals in Pascal. In general, a real constant in Pascal has one of the following forms:

$$m.n, \qquad m\mathbf{e}q, \qquad m.n\mathbf{e}q,$$

where m, q are signed or unsigned integers and n is an unsigned integer. Let Σ comprise the digits 0–9, the decimal point '.', the $+$ and $-$ signs, and the **e** for scientific notation. Let the set V of variables be $\{S, M, N, D\}$ and let the set P of the productions be:

$$\begin{array}{rcl} S & \to & M.N | M\mathbf{e}M | M.N\mathbf{e}M \\ M & \to & N | +N | -N \\ N & \to & DN | D \\ D & \to & 0|1|2|3|4|5|7|8|9 \end{array}$$

Then the grammar generates all valid Pascal real values (allowing redundant leading 0s). For instance, the value 12.3**e**-4 can be derived via

$$S \Rightarrow \underline{M}.N\mathbf{e}M \Rightarrow \underline{N}.N\mathbf{e}M \Rightarrow \underline{D}N.N\mathbf{e}M \Rightarrow 1\underline{N}.N\mathbf{e}M \Rightarrow 1\underline{D}.N\mathbf{e}M \Rightarrow$$

$$12.\underline{N}\mathbf{e}M \Rightarrow 12.\underline{D}\mathbf{e}M \Rightarrow 12.3\mathbf{e}\underline{M} \Rightarrow 12.3\mathbf{e} - \underline{N} \Rightarrow 12.3\mathbf{e}\text{-}\underline{D} \Rightarrow 12.3\mathbf{e}\text{-}4$$

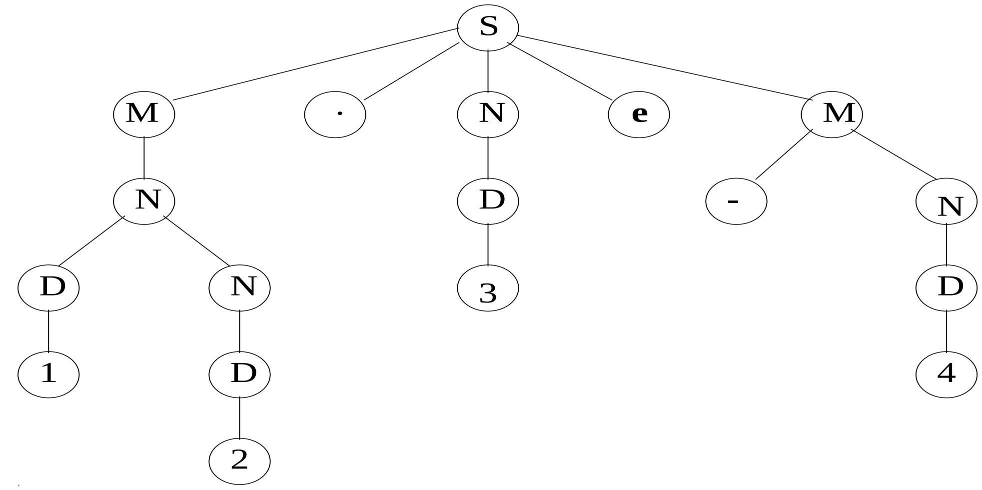

Figure 1: The derivation tree for 12.3**e** − 4.

Perhaps the most natural representation of derivations in a context-free grammar is a **derivation tree** or a **parse tree**. Every leaf of such a tree corresponds to a terminal (or to ϵ), and every internal node corresponds to a nonterminal. If A is an internal node with children $B_1, \ldots, B_n$, ordered from left to right, then $A \to B_1 \cdots B_n$ must be a production. The concatenation of all leaves from left to right yields the string being derived. For example, the derivation tree corresponding to the above derivation of 12.3**e**-4 is given in Figure 1. Such a tree also makes possible the extraction of the parts 12, 3 and -4, which are useful in the storage of the real value in a computer memory.

Definition 4.1 A context-free grammar G is **ambiguous** if there is a string $x \in L(G)$ that has two distinct derivation trees. Otherwise G is **unambiguous**.

Unambiguity is a very desirable property because it promises a unique interpretation of each sentence in the language. It is not hard to see that the grammar G_6 for Pascal real values and the grammar G_1 defined in Example 2.6 are both unambiguous. The following example shows an ambiguous grammar.

Example 4.2 Consider a grammar G_7 for all valid arithmetic expressions that are composed of unsigned positive integers and symbols $+, *, (,)$. For convenience, let us use the symbol **n** to denote any unsigned positive integer—it is treated as a terminal. This grammar has the productions

$$S \to T + S | S + T | T$$

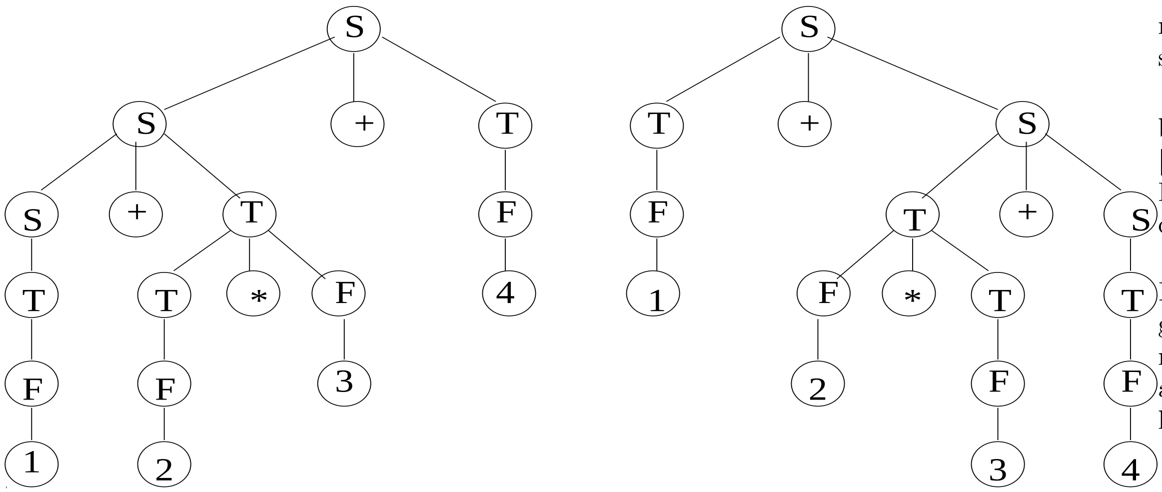

Figure 2: Different derivation trees for the expression $1 + 2 * 3 + 4$.

$$\begin{array}{rcl} T & \rightarrow & F * T | T * F | F \\ F & \rightarrow & \mathbf{n} | (S) \end{array}$$

Two possible different derivation trees for the expression $1 + 2 * 3 + 4$ are shown in Figure 2. Thus G_7 is ambiguous. The left tree means that the first addition should be done before the second addition, while the right tree says the opposite.

Although in the above example different derivations/interpretations of any expression always result in the same value because the operations addition and multiplication are associative, there are situations where the difference in the derivation can affect the final outcome. Actually, the grammar G_7 can be made unambiguous by removing the redundant productions $S \rightarrow T + S$ and $T \rightarrow F * T$. This corresponds to the convention that a sequence of consecutive additions or multiplications is always evaluated from left to right. Deleting the two productions does not change the language of strings generated by G_7, but it does fix unique interpretations of those strings.

It is worth noting that there are context-free languages that cannot be generated by any unambiguous context-free grammar. Such languages are said to be *inherently ambiguous*. An example taken from [Hopcroft and Ullman, 1979] (where this fact is proved) is

$$\{0^m 1^m 2^n 3^n | m, n > 0\} \cup \{0^m 1^n 2^n 3^m | m, n > 0\}.$$

The reason is that every context-free grammar G must yield two parse trees for some strings of the form $x = 0^n 1^n 2^n 3^n$, where one tree intuitively expresses that x is a

member of the first set of the union, and the other tree expresses that x is in the second set.

We end this section by presenting an efficient algorithm for the membership problem for context-free grammars, following the treatment in [Hopcroft and Ullman, 1979]. The algorithm is due to Cocke, Younger, and Kasami, and is often called the CYK algorithm. Let $G = (\Sigma, V, S, P)$ be a context-free grammar in Chomsky normal form.

Example 4.3 If we use the algorithm in [Hopcroft and Ullman, 1979] to convert the grammar G_7 from Example 4.2 into Chomsky normal form, we are led to introduce new "alias variables" A, B, C, D for the operators and parentheses, and "helper variables" S_1, T_1, T_2, F_1, F_2 to break up the productions in G_7 with right-hand-sides of length > 2 into length-2 pieces. The resulting grammar is:

$$\begin{array}{rcl} S & \rightarrow & T_1 S | S T_2 | F_1 T | T F_2 | C S_1 | \mathbf{n} \\ T_1 & \rightarrow & T A \\ T_2 & \rightarrow & A T \\ T & \rightarrow & F_1 T | T F_2 | C S_1 | \mathbf{n} \\ F_1 & \rightarrow & F B \\ F_2 & \rightarrow & B F \\ F & \rightarrow & \mathbf{n} | C S_1 \\ S_1 & \rightarrow & S D \\ A & \rightarrow & + \\ B & \rightarrow & * \\ C & \rightarrow & (\\ D & \rightarrow &) \end{array}$$

While this grammar is much less intuitive to read than G_7, having it in Chomsky normal form facilitates the description and operation of the CYK algorithm.

Now suppose that $x = a_1 \cdots a_n$ is a string of n terminals that we want to test for membership in $L(G)$. The basic idea of the CYK algorithm is a form of *dynamic programming*. For each pair i, j, where $1 \leq i \leq j \leq n$, define a set $X_{i,j} \subseteq V$ by

$$X_{i,j} = \{A | A \Rightarrow^* a_i \cdots a_j\}.$$

Then $x \in L(G)$ if and only if $S \in X_{1,n}$. The sets $X_{i,j}$ can be computed inductively in ascending order of $j - i$. It is easy to figure out $X_{i,i}$ for each i since $X_{i,i} = \{A | A \rightarrow a_i \in P\}$. Suppose that we have computed all $X_{i,j}$ where $j - i < d$ for some $d > 0$. To compute a set $X_{i,j}$, where $j - i = d$, we just have to find all the nonterminals A such that there exist some nonterminals B and C satisfying $A \rightarrow BC \in P$ and for some

Table 1: An example execution of the CYK algorithm.

		0	0	0	1	1	1
		$j \rightarrow$					
		1	2	3	4	5	6
	1	O					S
	2		O			S	T
i	3			O	S	T	
$\downarrow$	4				I		
	5					I	
	6						I

k, $i \leq k < j$, $B \in X_{i,k}$ and $C \in X_{k+1,j}$. A rigorous description of the algorithm in a Pascal-style pseudocode is given below.

Algorithm CYK($x = a_1 \cdots a_n$)

1. **for** $i \leftarrow 1$ **to** n **do**
2. $\quad X_{i,i} \leftarrow \{A | A \rightarrow a_i \in P\}$;
3. **for** $d \leftarrow 1$ **to** $n-1$ **do**
4. $\quad$ **for** $i \leftarrow 1$ **to** $n-d$ **do**
5. $\quad\quad X_{i,i+d} \leftarrow \emptyset$;
6. $\quad$ **for** $t \leftarrow 0$ **to** $d-1$ **do**
7. $\quad\quad\quad X_{i,i+d} \leftarrow X_{i,i+d} \cup \{A | A \rightarrow BC \in P$ for some $B \in X_{i,i+t}$ and $C \in X_{i+t+1,i+d}\}$;

Table 1 shows the sets $X_{i,j}$ for the grammar G_1 and the string $x = 000111$. In this run it happens that every $X_{i,j}$ is either empty or a singleton. The computation proceeds from the main diagonal toward the upper-right corner.

We now analyze the asymptotic time complexity of the CYK algorithm . Step 2 is executed n times. Step 5 is executed $\sum_{d=1}^{n-1} n - d = (n-1)(n-1+n-(n-1))/2 = n(n-1)/2 = O(n^2)$ times. Step 7 is repeated for $\sum_{d=1}^{n-1} d(n-d) = O(n^3)$ times. Therefore, the algorithm requires asymptotically $O(n^3)$ time to decide the membership of a string length n in $L(G)$, for any grammar G in Chomsky normal form.

5 More Efficient Parsing for Context-free Grammars

The CYK algorithm presented in the last section can be easily extended to solve the parsing problem for context-free grammars: In step 7, we also record a production $A \rightarrow BC$ and the corresponding value of t for any nonterminal A that gets added to $X_{i,i+d}$. Thus a derivation tree for x can be constructed by starting from the nonterminal S in $X_{1,n}$ and repeatedly applying the productions recorded for appropriate nonterminals in appropriate sets $X_{i,j}$. However, the cubic running time of this algorithm is generally too high for parsing applications. In practice, with compilation modules thousands of lines long, people seek grammars in other forms besides Chomsky's that permit parsing in linear or nearly-linear time.

Before we present some of these forms, we discuss parsing strategies in general. Parsing algorithms fall into two basic types, called **top-down parsers** and **bottom-up parsers**. As indicated by their names, a top-down parser builds derivation trees from the top (root) to the bottom (leaves), while a bottom-up parser starts from the leaves and works up to the root. Although neither method is good for handling all context-free grammars, each provides efficient parsing for many important subclasses of the context-free grammars, including those used in most programming languages.

We will only consider unambiguous grammars. To simplify the description of the parsers, we will assume that each string to be parsed ends with a special delimiter $ that does not appear anywhere else in the string. This assumption makes the detection of the end of the string easy in a left-to-right scan. The assumption does not put any serious restriction on the range of languages that can be parsed—the $ is just like the end-of-file marker in a real input file. The following definition will be useful.

Definition 5.1 A derivation from a sentential form to another is said to be **leftmost** (or **rightmost**) if at each step the leftmost (or rightmost, respectively) nonterminal is replaced.

For example, Example 4.3 gave a leftmost derivation of the word 12.3**e**-4 in the grammar G_6. For a given word x, leftmost derivations are in 1-1 correspondence with derivation trees, since we can find the leftmost derivation specified by a derivation tree by tracing the tree down from the root going from left to right. Rightmost derivations are likewise in 1-1 correspondence with derivation trees. Hence in an unambiguous context-free grammar, every derivable string has a unique leftmost derivation and a unique rightmost derivation. The parsing methods considered next find one or the other.

5.1 Top-down Parsing

An important member of the top-down parsers is the **LL parser** (see [Aho, Sethi and Ullman, 1985, Drobot, 1989]). Here, the first "L" means scanning the input from left to right, and the second means leftmost derivation. In other words, for any input string x, the parser intends to find the sequence of productions used in the leftmost derivation of x.

Let $G = (\Sigma, V, S, P)$ be a context-free grammar. A *parsing table* T for G has rows indexed by members of V and columns indexed by members of Σ and \$. Each entry $T[A, c]$ is either blank or contains one or more productions of the form $A \to \alpha$. Here we will suppose that G allows the construction of a parsing table T such that every non-blank entry $T[A, c]$ contains only *one* production. Then the LL parser for G is a device very similar to a pushdown automaton as described in Chapter 30. The parser has an input buffer, a pushdown stack, a parsing table, and an output stream. The input buffer contains the string to be parsed followed by the delimiter \$. The stack contains a sequence of terminals or nonterminals, with another delimiter # that marks the bottom of the stack. Initially, the input pointer points to the first symbol of the input string, and the stack contains the start nonterminal S on top of #. Figure **??** illustrates schematically the components of the parser. As usual, the input pointer will only move to the right, while the stack pointer is allowed to move up and down.

The parser is controlled by an algorithm that behaves as follows. At any instant of time, the algorithm considers the symbol X on top of the stack and the current input symbol c pointed by the input pointer, and makes one of the following moves.

1. If X is a nonterminal, the algorithm consults the entry $T[X, a]$ of the parsing table T. If the entry is blank, the parser halts and states that the input string x is not in the language $L(G)$. If not, the entry is a production of the form $X \to Y_1 \cdots Y_k$. Then the algorithm replaces the top stack symbol X with the string $Y_1 \cdots Y_k$ (with Y_1 on top), and outputs the production.

2. If X is a terminal, X is compared with c. If $X = c$, the algorithm pops X off the stack and shifts the input pointer to the next input symbol. Otherwise, the algorithm halts and states that $x \notin L(G)$.

3. If $X = \#$, then provided $c = \$$, the algorithm halts and declares the successful completion of parsing. Otherwise the algorithm halts and states that $x \notin L(G)$.

Intuitively, the parser reconstructs the derivation of a string $x = a_1 \cdots a_n$ as follows. Suppose that the leftmost derivation of x is

$$S = \gamma_0 \Rightarrow \gamma_1 \Rightarrow \cdots \Rightarrow \gamma_i \Rightarrow \gamma_{i+1} \Rightarrow \cdots \Rightarrow \gamma_m = x,$$

Table 2: An LL parsing table for grammar G_8.

NONTERMINAL	INPUT SYMBOL					
	n	$+$	$*$	(	)	\$
S	$S \to TS'$			$S \to TS'$		
S'		$S' \to +S$			$S' \to \epsilon$	$S' \to \epsilon$
T	$T \to FT'$			$T \to FT'$		
T'		$T' \to \epsilon$	$T' \to *T$		$T' \to \epsilon$	$T' \to \epsilon$
F	$F \to \mathbf{n}$			$F \to (S)$		

where each γ_j is a sentential form. Suppose, moreover, that the derivation step $\gamma_i \Rightarrow \gamma_{i+1}$ is the result of applying a production $X \to Y_1 \cdots Y_k$. This means that $\gamma_i = \alpha X \beta$ for some string α of terminals and sentential form β. Since no subsequent derivation will change α, this string must match a leading substring $a_1 \cdots a_j$ of x for some j. In other words, $\gamma_i = a_1 \cdots a_j X \beta$ and $\gamma_{i+1} = a_1 \cdots a_j Y_1 \cdots Y_k \beta$. Suppose that the parser has successfully reconstructed the derivation steps up to γ_i. To complete the derivation, the parser must transform the tail end of γ_i into $a_{j+1} \cdots a_n$. Thus, it keeps the string $X\beta$ on the stack and repeatedly replaces the top stack symbol (i.e., replaces the leftmost nonterminal) until a_{j+1} appears on top. At this point, a_{j+1} is removed from the stack, and the remainder of the stack must be transformed to match $a_{j+2} \cdots a_n$. The procedure is repeated until all the input symbols are matched.

The following example illustrates the parsing table for a simple context-free grammar, and how the parser operates.

EXAMPLE 5.1 Consider again the language of valid arithmetic expressions from Example 4.2, where an ambiguous grammar G_7 was given that could be made unambiguous by removing two productions. Let us remove the ambiguity in a different way. The new grammar is called G_8 and has the following productions

$$\begin{array}{rcl} S & \to & TS' \\ S' & \to & +S \mid \epsilon \\ T & \to & FT' \\ T' & \to & *T \mid \epsilon \\ F & \to & \mathbf{n} \mid (S) \end{array}$$

It is easy to see that grammar G_8 is unambiguous. A parsing table for this grammar is shown in Table 2. We will discuss how such a table can be constructed shortly.

To demonstrate how the parser operates, consider the input string $(\mathbf{n} + \mathbf{n}) * \mathbf{n}$. Table 3 shows the content of the stack, the remaining input symbols, and the output after each step. If we trace the actions of the parser carefully, we see that the sequence of productions it outputs constitutes the leftmost derivation of $(\mathbf{n} + \mathbf{n}) * \mathbf{n}$.

Table 3: The steps in the LL parsing of $(\mathbf{n}+\mathbf{n})*\mathbf{n}$.

STACK	INPUT	OUTPUT
$S\#$	$(\mathbf{n}+\mathbf{n})*\mathbf{n}\$$	
$TS'\#$	$(\mathbf{n}+\mathbf{n})*\mathbf{n}\$$	$S \to TS'$
$FT'S'\#$	$(\mathbf{n}+\mathbf{n})*\mathbf{n}\$$	$T \to FT'$
$(S)T'S'\#$	$(\mathbf{n}+\mathbf{n})*\mathbf{n}\$$	$F \to (S)$
$S)T'S'\#$	$\mathbf{n}+\mathbf{n})*\mathbf{n}\$$	
$TS')T'S'\#$	$\mathbf{n}+\mathbf{n})*\mathbf{n}\$$	$S \to TS'$
$FT'S')T'S'\#$	$\mathbf{n}+\mathbf{n})*\mathbf{n}\$$	$T \to FT'$
$\mathbf{n}T'S')T'S'\#$	$\mathbf{n}+\mathbf{n})*\mathbf{n}\$$	$F \to \mathbf{n}$
$T'S')T'S'\#$	$+\mathbf{n})*\mathbf{n}\$$	
$S')T'S'\#$	$+\mathbf{n})*\mathbf{n}\$$	$T' \to \epsilon$
$+S)T'S'\#$	$+\mathbf{n})*\mathbf{n}\$$	$S' \to +S$
$S)T'S'\#$	$\mathbf{n})*\mathbf{n}\$$	
$TS')T'S'\#$	$\mathbf{n})*\mathbf{n}\$$	$S \to TS'$
$FT'S')T'S'\#$	$\mathbf{n})*\mathbf{n}\$$	$T \to FT'$
$\mathbf{n}T'S')T'S'\#$	$\mathbf{n})*\mathbf{n}\$$	$F \to \mathbf{n}$
$T'S')T'S'\#$	$)*\mathbf{n}\$$	
$S')T'S'\#$	$)*\mathbf{n}\$$	$T' \to \epsilon$
$)T'S'\#$	$)*\mathbf{n}\$$	$T' \to \epsilon$
$T'S'\#$	$*\mathbf{n}\$$	
$*TS'\#$	$*\mathbf{n}\$$	$T' \to *T$
$TS'\#$	$\mathbf{n}\$$	
$FT'S'\#$	$\mathbf{n}\$$	$T \to FT'$
$\mathbf{n}T'S'\#$	$\mathbf{n}\$$	$F \to \mathbf{n}$
$T'S'\#$	$\$$	$T' \to \epsilon$
$S'\#$	$\$$	$S' \to \epsilon$
$\#$	$\$$	

Now we turn to the question of how to construct an LL parser for a given grammar $G = (\Sigma, V, S, P)$. It suffices to show how to compute the entries $T[A, c]$, where $A \in V$ and $c \in \Sigma \cup \{\$\}$. We first need to introduce two functions $FIRST(\alpha)$ and $FOLLOW(A)$. The former maps a sentential form to a terminal or ϵ, and the latter maps a nonterminal to a terminal or \$.

DEFINITION 5.2 For each sentential form $\alpha \in \{\Sigma \cup V\}^*$, and for each nonterminal $A \in V$,

$$\begin{aligned} FIRST(\alpha) &= \{c \in \Sigma \mid \text{for some } \beta \in \{\Sigma \cup V\}^*,\ \alpha \Rightarrow^* c\beta\} \cup \{\epsilon \mid \alpha \Rightarrow^* \epsilon\} \\ FOLLOW(A) &= \{c \in \Sigma \mid \text{for some } \alpha, \beta \in \{\Sigma \cup V\}^*,\ S \Rightarrow^* \alpha A c \beta\} \\ &\cup \{\$ \mid \text{for some } \alpha \in \{\Sigma \cup V\}^*,\ S \Rightarrow^* \alpha A\}. \end{aligned}$$

Intuitively, for any sentential form α, $FIRST(\alpha)$ consists of all the terminals that appear as the first symbol of some sentential form derivable from α. The empty string ϵ is included in $FIRST(\alpha)$ as a special case if α derives ϵ. On the other hand, for any nonterminal A, $FOLLOW(A)$ consists of all the terminals that immediately follow an occurrence of A in some sentential form derivable from the start symbol S. The end delimiter \$ is included in $FOLLOW(A)$ as a special case if A appears at the end of some sentential form derivable from S.

Algorithms for computing the $FIRST()$ and $FOLLOW()$ functions are fairly straightforward and can be found in [Aho, Sethi and Ullman, 1985, Drobot, 1989]. It turns out that to construct the parsing table for a grammar G, we only need to know the values of $FIRST(\alpha)$ for those sentential forms α appearing on the right-hand sides of the productions in G.

EXAMPLE 5.2 The following illustrate the functions $FIRST(\alpha)$ and $FOLLOW(A)$ for the grammar G_8 described in the above example. For the former, only those sentential forms appearing on the right-hand sides of the productions in G_8 are considered.

$$\begin{aligned} FIRST(TS') &= \{(, \mathbf{n}\} \\ FIRST(+S) &= \{+\} \\ FIRST(FT') &= \{(, \mathbf{n}\} \\ FIRST(*T) &= \{*\} \\ FIRST((S)) &= \{(\} \\ FIRST(\mathbf{n}) &= \{\mathbf{n}\} \\ FIRST(\epsilon) &= \{\epsilon\} \\ FOLLOW(S) &= \{), \$\} \end{aligned}$$

$$\begin{aligned} FOLLOW(S') &= \{), \$\} \\ FOLLOW(T) &= \{+,), \$\} \\ FOLLOW(T') &= \{+,), \$\} \\ FOLLOW(F') &= \{*, +,), \$\} \end{aligned}$$

Given the functions $FIRST(\alpha)$ and $FOLLOW(A)$ for a grammar G, we can easily construct the LL parsing table $T[A, c]$ for G. The basic idea is as follows. Suppose that $A \to \alpha$ is a production and $c \in FIRST(\alpha)$. Then, the parser will replace A with α when A is on top of the stack and c is the current input symbol. The only complication occurs when α may derive ϵ. In this case, the parser should still replace A with α if the current input symbol is a member of $FOLLOW(A)$. The detailed algorithm is given below.

Algorithm LL-Parsing-Table($G = (\Sigma, V, S, P)$)

Initialize each entry of the table to blank.
for each production $A \to \alpha$ in P **do**
    for each terminal $a \in FIRST(\alpha)$ **do**
        add $A \to \alpha$ to $T[A, a]$;
    if $\epsilon \in FIRST(\alpha)$ **then**
        for each terminal or delimiter $a \in FOLLOW(A)$ **do**
            add $A \to \alpha$ to $T[A, a]$;

The above algorithm can be applied to any context-free grammar to produce a parsing table. However, for some grammars the table may have entries containing multiple productions. Multiply-defined entries in a parsing table, however, would present our parsing algorithm with an unwelcome choice. It would be possible for it to make a wrong choice and incorrectly report a string as not being derivable, and backtracking to the last choice to try another would blow up the running time unacceptably.

Example 5.3 Recall that we could make the grammar G_7 of Example 4.2 unambiguous by deleting two unnecessary productions. The resulting grammar, which we call G_9, has the following productions:

$$\begin{aligned} S &\to S + T | T \\ T &\to T * F | F \\ F &\to \mathbf{n} | (S) \end{aligned}$$

It is easy to see that both $FIRST(S + T)$ and $FIRST(T)$ contain the terminal **n**. Hence, the entry $T[S, \mathbf{n}]$ of the parsing table is multiply defined, so this table is not well-conditioned for LL parsing.

A context-free grammar whose parsing table has no multiply-defined entries is called an **LL(1) grammar**. Here, the "1" signifies the fact that the LL parser uses one input symbol of lookahead to decide its next move. For example, G_8 is an LL(1) grammar, while G_9 is not. It is easy to show that our LL parser runs in linear time for any LL(1) grammar.

What can we do for grammars that are not LL(1), such as G_9? The first idea is to extend the LL parser to use more input symbols of lookahead. In other words, we will allow the parser to see the next several input symbols before it makes a decision. For one more symbol of lookahead, this requires expanding the parsing table to have a column for every *pair* of symbols in Σ (plus \$ as a possible second symbol), but so doing may separate and/or eliminate multiply-defined entries in the original parsing table. The $FIRST()$ and $FOLLOW()$ functions have to be modified to take two (or more) lookahead symbols into consideration. For any constant $k > 1$, a grammar is said to be an **LL(k) grammar** if its parsing table using k lookahead symbols has no multiply defined entries. For example, the grammar G_1 given in Example 2.6 is not LL(1), but it is LL(2).

Although LL(k) grammars form a larger class than LL(1) grammars, there are still grammars that are not LL(k) for any constant k. The grammar G_7 and G_9 are examples. The texts [Aho, Sethi and Ullman, 1985, Drobot, 1989] provide several techniques for dealing with non-LL(k) grammars, such as grammar transformations and backtracking. When backtracking is used, the parsing process is often called *recursive-descent parsing*, and can be very time consuming due to the use of many recursive calls.

5.2 Bottom-up Parsing

The most popular bottom-up parsing technique is **LR parsing**. Here, the "L" again means scanning the input from left to right, while the "R" means constructing the rightmost derivation. For any input string x, the LR parser scans x from left to right and tries to find the *reverse* of the sequence of productions used in the rightmost derivation of x. It turns out that in bottom-up parsing rightmost derivations are easier to deal with than leftmost derivations. LR parsing is especially attractive in practice for many reasons summarized in [Aho, Sethi and Ullman, 1985]: (i) it can handle virtually all programming language constructs; (ii) it has very efficient implementations; (iii) it is more powerful than LL parsing; and (iv) it detects syntactic errors quickly. The principal drawback of the method is that constructing an LR parser is very involved. Fortunately, there exist efficient algorithms that can automatically generate LR parsers from certain context-free grammars. Because of space limitations, we describe only the operation of an LR parser here, and refer the reader to [Aho, Sethi and Ullman, 1985] for the construction of such a parser.

Similar to an LL parser, an LR parser has an input buffer, a pushdown stack, a parsing table, and an output stream, and is controlled by an algorithm that is the same for all LR parsers. The input string is again assumed to have an end delimiter \$. At any time during parsing, the stack stores a string of the form $q_m X_m q_{m-1} \cdots X_1 q_0$ (with q_0 at bottom), where each X_i is a grammar symbol (i.e., a terminal or nonterminal of the grammar involved) and q_i is a *state* symbol. The number of distinct states is finite, and each state symbol intends to summarize the information contained in the stack below it. The combination of the state on top of the stack and the current input symbol are used to index the parsing table and determine the move of the parser. It will be seen that the state symbols subsume all information in the grammar symbols, and a real parser omits the latter. However, we retain the grammar symbols $X_1, \ldots, X_m$ to make our illustration easier to follow, and for consistency with previous examples.

The parsing table consists of two parts: a parsing action function $ACTION(q, c)$, which maps a state and an input symbol to a move, and a function $GOTO(q, X)$, which maps a state and a grammar symbol to a state. For each state q and each input symbol c, the value of the function $ACTION(q, c)$ can be one of the following:

1. *shift*,
2. *reduce by* $A \to \alpha$, where $A \to \alpha$ is a production in the grammar,
3. *accept*, and
4. *blank*.

The algorithm controlling the LR parser operates as follows. Suppose that the state on top of the stack is q and the current input symbol is c. It consults $ACTION(q, c)$ and makes one of the four types of moves as below.

1. If $ACTION(q, c) =$ *shift*, the parser pushes the string $GOTO(q, c)c$ on the stack and shifts its input pointer to the next input symbol.
2. If $ACTION(q, c) =$ *reduce by* $A \to \alpha$, the parser applies the production $A \to \alpha$ as follows. Let $k = |\alpha|$, and let the current stack content be $q_m X_m q_{m-1} \cdots X_1 q_0$. The parser first pops the top $2k$ symbols $q_m, X_m, \ldots, q_{m-k+1}, X_{m-k+1}$ off the stack. It then consults $GOTO(q_{m-k}, A)$ and pushes the string $GOTO(q_{m-k}, A)A$ onto the stack, resulting in a stack with content $GOTO(q_{m-k}, A)Aq_{m-k}X_{m-k} \cdots X_1 q_0$. The parser also outputs the production $A \to \alpha$.

   It is always guaranteed in the above that $X_{m-k+1} \cdots X_m = \alpha$.
3. If $ACTION(q, a) =$ *accept*, the parser successfully terminates.
4. If $ACTION(q, a) =$ *blank*, the parser terminates and declares that the input string is not a member of the language.

Intuitively, the LR parser reconstructs the rightmost derivation of a string $x = a_1 \cdots a_n$ as follows. Suppose that the rightmost derivation of x is

$$S = \gamma_0 \Rightarrow \gamma_1 \Rightarrow \cdots \Rightarrow \gamma_i \Rightarrow \gamma_{i+1} \Rightarrow \cdots \Rightarrow \gamma_m = x,$$

where each γ_j is a sentential form. Furthermore, suppose that the derivation step $\gamma_i \Rightarrow \gamma_{i+1}$ is the result of applying a production $A \to Y_1 \cdots Y_k$. This means that $\gamma_i = \alpha A z$ for some sentential form $\alpha = X_1 \cdots X_t$ and string z of terminals, and $\gamma_{i+1} = \alpha Y_1 \cdots Y_k z = X_1 \cdots X_t Y_1 \cdots Y_k z$. Since no subsequent derivation will change z, this string must match a trailing substring $a_j \cdots a_n$ of x for some j. In other words, $\gamma_i = X_1 \cdots X_t A a_j \cdots a_n$ and $\gamma_{i+1} = X_1 \cdots X_t Y_1 \cdots Y_k a_j \cdots a_n$.

Suppose that the parser has successfully reconstructed the derivation steps in reverse from γ_m back to γ_{i+1}. At this point, the stack must be holding a string of the form

$$q_{t+h} Y_h \cdots q_{t+1} Y_1 q_t X_t \cdots q_1 X_1 q_0,$$

where $h \leq k$ and $q_0, q_1, \ldots q_{t+h}$ are some states, and the input pointer is pointing at a_{j+h-k}. Moreover, it must be that $Y_{h+1} = a_{j+h-k}, \ldots, Y_k = a_{j-1}$. To recover γ_i, the parser consults the state q_{t+h} on top of stack and the current input symbol a_{j+h-k}. It then shifts the $h - k$ input symbols $a_{j+h-k}, \ldots, a_{j-1}$ and $h - k$ appropriate state symbols onto the stack. It also advances the input pointer to a_j. Then, the parser *reduces* the string $Y_1 \cdots Y_k$ to the nonterminal A by replacing the top $2k$ stack symbols with A and an appropriate state symbol.

The above shift-and-reduce process is repeated until the sentential form $\gamma_0 = S$ is obtained. For this reason, the LR parser is sometimes called a *shift-reduce parser*.

Clearly, the state symbols stored on the stack play a key role in dictating the actions of the parser. Below we first give an example of LR parsing tables and show exactly how the parser operates on a specific input. Then we will briefly sketch how the states are chosen for a grammar and what they represent.

Example 5.4 Consider again the unambiguous grammar G_9 given in Example 5.3. For convenience, let us number the productions as follows.

$$\begin{array}{rlcl}
(1) & S & \to & S + T \\
(2) & S & \to & T \\
(3) & T & \to & T * F \\
(4) & T & \to & F \\
(5) & F & \to & \mathbf{n} \\
(6) & F & \to & (S)
\end{array}$$

Table 4: The function $ACTION(q, c)$ for the unambiguous grammar G_9.

STATE	**n**	+	$*$	(	)	\$
0	shf			shf		
1		shf				acc
2		p2	shf		p2	p2
3		p4	p4		p4	p4
4	shf			shf		
5		p6	p6		p6	p6
6	shf			shf		
7	shf			shf		
8		shf			shf	
9		p1	shf		p1	p1
10		p3	p3		p3	p3
11		p5	p5		p5	p5

Tables 4 and 5 illustrate the functions $ACTION(q, c)$ and $GOTO(q, X)$ for the grammar. In the first table, shf means shift, pi means reduce by production i, acc means accept, and blank means reject. The states are numbered $0, 1, \ldots, 11$.

Now we demonstrate how the parser operates on the string $(\mathbf{n} + \mathbf{n}) * \mathbf{n}$. Table 6 shows the content of the stack, the remaining input symbols, and the output after each step. It is easy to see that the reverse sequence of the productions in the reduce steps constitute the rightmost derivation of $(\mathbf{n} + \mathbf{n}) * \mathbf{n}$.

There are several techniques for constructing an LR parsing table, such as simple-LR (SLR), canonical-LR, and lookahead-LR (LALR), as described by [Aho, Sethi and Ullman, 1985]. In general, these techniques all use states that are sets of *items* of the form $A \to \alpha \cdot \beta$, where $A \to \alpha\beta$ is a production and the $\cdot$ marks a place in the right-hand side. Such items are commonly known as the *LR items*. Each item expresses the assertion that the part α has already been obtained by previous shift/reduce steps and pushed on the stack, and the part β is expected to be obtainable from the next few input symbols by some shift/reduce steps. Since at any given time the parser may not be able to predict what input symbols should follow, it has to maintain a set of LR items to deal with all possibilities.

Again, not all context-free grammars have effective LR parsers. For example, the grammar with productions

$$S \quad \to \quad 0S0|1S1|0|1|\epsilon$$

cannot be handled by LR parsing. This grammar generates the set of all palindromes. The grammars that have effective LR parsers are called *LR grammars*. In fact, there

Table 5: The function $GOTO(q, X)$ for the unambiguous grammar G_9.

STATE	**n**	+	$*$	(	)	\$	S	T	F
0	5			4			1	2	3
1		6							
2			7						
3									
4	5			4			8	2	3
5									
6	5			4				9	3
7	5			4					10
8		8			11				
9		7							
10									
11									

Table 6: The steps in the LR parsing of $(\mathbf{n} + \mathbf{n}) * \mathbf{n}$.

STACK	INPUT	ACTION
0	$(\mathbf{n} + \mathbf{n}) * \mathbf{n}\$$	shift
$4(0$	$\mathbf{n} + \mathbf{n}) * \mathbf{n}\$$	shift
$5\mathbf{n}4(0$	$+\mathbf{n}) * \mathbf{n}\$$	reduce by $F \to \mathbf{n}$
$3F4(0$	$+\mathbf{n}) * \mathbf{n}\$$	reduce by $T \to F$
$2T4(0$	$+\mathbf{n}) * \mathbf{n}\$$	reduce by $S \to T$
$8S4(0$	$+\mathbf{n}) * \mathbf{n}\$$	shift
$6 + 8S4(0$	$\mathbf{n}) * \mathbf{n}\$$	shift
$5\mathbf{n}6 + 8S4(0$	$) * \mathbf{n}\$$	reduce by $F \to \mathbf{n}$
$3F6 + 8S4(0$	$) * \mathbf{n}\$$	reduce by $T \to F$
$9T6 + 8S4(0$	$) * \mathbf{n}\$$	reduce by $S \to S + T$
$8S4(0$	$) * \mathbf{n}\$$	shift
$11)8S4(0$	$*\mathbf{n}\$$	reduce by $F \to (S)$
$3F0$	$*\mathbf{n}\$$	reduce by $T \to F$
$2T0$	$*\mathbf{n}\$$	shift
$7 * 2T0$	$\mathbf{n}\$$	shift
$5\mathbf{n}7 * 2T0$	\$	reduce by $F \to \mathbf{n}$
$10F7 * 2T0$	\$	reduce by $T \to T * F$
$2T0$	\$	reduce by $S \to T$
$1S0$	\$	accept

are context-free languages that cannot be represented by any LR grammars. The set of palindromes is one such language.

6 Defining Terms

ambiguous context-free grammar: a context-free grammar in which some derivable terminal strings have two distinct derivation trees.

bottom-up parsing: a process of building a derivation tree from the leaves up to the root.

Chomsky normal form: a form of context-free grammar in which every rule has the form $A \to BC$ or $A \to a$, where A, B, C are nonterminals and a is a terminal.

context-free grammar: a grammar whose rules have the form $A \to \beta$, where A is a nonterminal and β is a string of nonterminals and terminals.

context-free language: a language that can be described by some context-free grammar.

context-sensitive grammar: a grammar whose rules have the form $\alpha \to \beta$, where α, β are strings of nonterminals and terminals, and $|\alpha| \leq |\beta|$.

context-sensitive language: a language that can be described by some context-sensitive grammar.

derivation or **parsing**: a sequence of applications of rules of a grammar that transforms the start symbol into a given terminal string or sentential form.

derivation tree or **parse tree**: a rooted, ordered tree that describes a particular derivation of a string with respect to some context-free grammar.

(formal) language: a set of strings over some fixed alphabet.

(formal) grammar: a description of some language, typically consisting of a set of terminals, a set of nonterminals, a distinguished nonterminal called the start symbol, and a set of rules (or productions) of the form $\alpha \to \beta$, which determine which substrings α of a sentential form can be replaced by some another string β.

leftmost (or **rightmost**) **derivation**: a derivation in which at each step, the leftmost (respectively, rightmost) nonterminal is rewritten.

LL parsing: a type of top-down parsing in which one reads the input from left to right in order to reconstruct a leftmost derivation.

LL(k) grammar: a context-free grammar whose LL(k) parsing table has no multiply-defined entries.

LL(k) parsing: an LL parsing that uses k symbols of lookahead.

LR parsing: a type of bottom-up parsing in which one reads the input from left to right in order to reconstruct a rightmost derivation in reverse order of steps.

LR grammar: a context-free grammar that has an effective LR parser.

membership problem (or **lexical analysis**): the problem or process of deciding whether a given string is generated by a given grammar.

parsing problem: the problem of reconstructing a derivation of a given input string in a given grammar.

regular expression: a description of some language using the operators union, concatenation, and Kleene closure.

regular language: a language that can be described by some regular expression, or equivalently, by some right-linear/regular grammar.

right-linear or **regular grammar**: a grammar whose rules have the form $A \to cB$, $A \to c$, or $A \to \epsilon$, where A, B are nonterminals, c is a terminal, and ϵ is the empty string.

sentential form: a string of terminals and nonterminals obtained at some step of a derivation in a grammar.

top-down parsing: a process of building derivation trees from the top (root) down to the bottom (leaves).

References

[Aho, Sethi and Ullman, 1985] Aho, A.V., Ullman, J.D. and Sethi, I. 1985. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley, Reading, MA.

[Angluin, 1980] Angluin, D. 1980. Finding patterns common to a set of strings. *Journal of Computer and System Sciences*. 21:46-62.

[Chomsky, 1956] Chomsky, N. 1956. Three models for the description of language. *IRE Trans. on Information Theory*. 2(2):113-124.

[Chomsky, 1963] Chomsky, N. 1963. Formal properties of grammars. In *Handbook of Mathematical Psychology* Vol. 2, 323-418. John Wiley and Sons, New York.

[Chomsky and Miller, 1958] Chomsky, N. and Miller, G. 1958. Finite-state languages. *Information and Control*. 1:91-112.

[Drobot, 1989] Drobot, V. 1989. *Formal Languages and Automata Theory*. Computer Science Press, Rockville, MD.

[Floyd and Beigel, 1994] Floyd, R.W. and Beigel, R. 1994. *The Language of Machines: an Introduction to Computability and Formal Languages*. Computer Science Press, New York.

[Gurari, 1989] Gurari, E. 1989. *An Introduction to the Theory of Computation*. Computer Science Press, Rockville, MD.

[Harel, 1992] Harel, D. 1992. *Algorithmics: The Spirit of Computing*. Addison-Wesley, Reading, MA.

[Harrison, 1978] Harrison, M. 1978. *Introduction to Formal Language Theory.* Addison-Wesley, Reading, MA.

[Hopcroft and Ullman, 1979] Hopcroft, J. and Ullman, J. 1979. *Introduction to Automata Theory, Languages and Computation.* Addison-Wesley, Reading, MA.

[Jiang et al., 1995] Jiang, T., Salomaa, A., Salomaa, K., and Yu, S. 1995. Decision problems for patterns. *Journal of Computer and System Sciences.* 50(1):53-63.

[Kleene, 1956] Kleene, S. 1956. Representation of events in nerve nets and finite automata. In *Automata Studies*, 3-41. Princeton University Press, NJ.

[Lind and Marcus, 1995] Lind, D. and Marcus, 1995 *Symbolic Dynamics*, Academic Press.

[Post, 1943] Post, E. 1943. Formal reductions of the general combinatorial decision problems. *Amer. J. Math.* 65:197-215.

[Salomaa, 1966] Salomaa, A. 1966. Two complete axiom systems for the algebra of regular events. *J. ACM.* 13(1):158-169.

[Searls, 1993] Searls, D. 1993. The computational linguistics of biological sequences. In *Artificial Intelligence and Molecular Biology.* L. Hunter (ed.), MIT Press, 1993, pp. 47-120.

[Wood, 1987] Wood, D. 1987. *Theory of Computation.* Harper and Row.

Further Information

The fundamentals of formal languages and grammars can be found in many text books including [Drobot, 1989, Floyd and Beigel, 1994, Gurari, 1989, Harel, 1992, Harrison, 1978, Hopcroft and Ullman, 1979, Wood, 1987]. The central focus of research in this area has been to find formal grammatical representations of languages that are very expressive and are yet easy to parse. The research results have greatly benefited many fields of computer science, including programming languages, compiler design, and natural language processing. The preceding chapter presents the machine model counterparts of regular grammars, context-free grammars, context-sensitive grammars, and unrestricted grammars, and the next chapter introduces the concepts of decidability and undecidability, which has a close relation to formal grammars. The following annual conferences present the leading research work in formal languages and grammars: International Colloquium on Automata, Languages and Programming (ICALP), ACM Annual Symposium on Theory of Computing (STOC), IEEE Symposium on the Foundations of Computer Science (FOCS), ACM Symposium on Principles of Programming Languages (POPL), Symposium on Theoretical Aspects of Computer Science (STACS), Mathematical Foundations of Computer Science (MFCS), Fundamentals of Computation Theory (FCT), Foundation of Software Technology and Theoretical Computer Science (FSTTCS), and Conference on

Developments in Language Theory (DLT). There are many related conferences, including Computational Learning Theory (COLT), Colloquium on Trees in Algebra and Programming (CAAP), and International Conference on Concurrency Theory (CONCUR), where either specific issues concerning formal grammars are considered or specialized grammatical systems are studied for a specific application area. We conclude with a list of major journals that publish papers in formal language theory: *Journal of the ACM*, *SIAM Journal on Computing*, *Journal of Computer and System Sciences*, *Information and Computation*, *Theory of Computing Systems* (formerly *Mathematical Systems Theory*), *Theoretical Computer Science*, *Information Processing Letters*, *International Journal of Foundations of Computer Science*, and *Acta Informatica*.