

# Chapter 2

## Syntax

### 2.1 Introduction

#### *2.1.1 Artificial and Formal Languages*

Many centuries after the spontaneous emergence of natural language for human communication, mankind has purposively constructed other communication systems and languages, to be called artificial, intended for very specific tasks. A few artificial languages, like the logical propositions of Aristotle or the music sheet notation of Guittone d'Arezzo, are very ancient, but their number has exploded with the invention of computers. Many of them are intended for man-machine communication, to instruct a programmable machine to do some task: to perform a computation, to prepare a document, to search a database, to control a robot, and so on. Other languages serve as interfaces between devices, e.g., Postscript is a language produced by a text processor commanding a printer.

Any designed language is artificial by definition, but not all artificial languages are formalized: thus a programming language like Java is formalized, but Esperanto, although designed by man, is not.

For a language to be formalized (or formal), the form of sentences (or syntax) and their meaning (or semantics) must be precisely and algorithmically defined. In other words, it should be possible for a computer to check that sentences are grammatically correct, and to determine their meaning.

Meaning is a difficult and controversial notion. For our purposes, the meaning of a sentence can be taken to be the translation to another language which is known to the computer or the operator. For instance, the meaning of a Java program is its translation to the machine language of the computer executing the program.

In this book the term *formal language* is used in a narrower sense that excludes semantics. In the field of syntax, a formal language is a mathematical structure, defined on top of an alphabet, by means of certain axiomatic rules

(formal grammar) or by using abstract machines such as the famous one due to A. Turing. The notions and methods of formal language are analogous to those used in number theory and in logic.

Thus formal language theory is only concerned with the form or syntax of sentences, not with meaning. A string (or text) is either valid or illegal, that is, it either belongs to the formal language or does not. Such theory makes a first important step towards the ultimate goal: the study of language translation and meaning, which will require additional methods.

### *2.1.2 Language Types*

A language in this book is a one-dimensional communication medium, made by sequences of symbolic elements of an alphabet, called terminal characters. Actually people often refer to language as other not textual communication media, which are more or less formalized by means of rules. Thus iconic languages focus on road traffic signs or video display icons. Musical language is concerned with sounds, rhythm, and harmony. Architects and designers of buildings and objects are interested in their spatial relations, which they describe as the language of design. Early child drawings are often considered as sentences of a pictorial language, which can be partially formalized in accordance with psychological theories. The formal approach to the syntax of this chapter has some interest for nontextual languages too.

Within computer science, the term language applies to a text made by a set of characters orderly written from, say, left to right. In addition the term is used to refer to other discrete structures, such as graphs, trees, or arrays of pixels describing a discrete picture. Formal language theories have been proposed and used to various degrees also for such nontextual languages<sup>1</sup>.

Reverting to the main stream of textual languages, a frequent request directed to the specialist is to define and specify an artificial language. The specification may have several uses: as a language reference manual for future users, as an official standard definition, or as a contractual document for compiler designers to ensure consistency of specification and implementation.

It is not an easy task to write a complete and rigorous definition of a language. Clearly the exhaustive approach, to list all possible sentences or phrases, is unfeasible because the possibilities are infinite, since the length of sentences is usually unbounded. As a native language speaker, a programmer is not constrained by any strict limit on the length of phrases to be written. The problem to represent an infinite number of cases by a finite description can be addressed by an enumeration procedure, as in logic. When executed, the procedure generates longer and longer sentences, in an unending process if the language to be modelled is not finite.

---

<sup>1</sup> Just two examples and references: tree languages [21] and picture (or two dimensional) languages [23, 15].

This chapter presents a simple and established manner to express the rules of the enumeration procedure in the form of rules of a generative grammar (or syntax).

### *2.1.3 Chapter Outline*

The chapter starts with the basic components of language theory: alphabet, string, and operations, such as concatenation and repetition, on strings and sets of strings.

The definition of the family of regular languages comes next.

Then the lists are introduced as a fundamental and pervasive syntax structure in all kinds of languages. From the exemplification of list variants, the idea of linguistic abstraction grows out naturally. This is a powerful reasoning tool to reduce the varieties of existing languages to a few paradigms.

After discussing the limits of regular languages, the chapter moves to context-free grammars. After the basic definitions the presentation focuses on structural properties, namely, equivalence, ambiguity, and recursion.

Exemplification continues with important linguistic paradigms such as: hierarchical lists, parenthesized structures, polish notations, and operator precedence expressions. Their combination produces the variety of forms to be found in artificial languages.

Then the classification of some common forms of ambiguity and corresponding remedies is offered as a practical guide for grammar designers.

Various transformations of rules (normal forms) are introduced, which should familiarize the reader with the modifications often needed for technical applications, to adjust a grammar without affecting the language it defines.

Returning to regular languages from the grammar perspective, the chapter evidences the greater descriptive capacity of context-free grammars.

The comparison of regular and context-free languages continues by considering the operations that may cause a language to exit or remain in one or the other family. Alphabetical transformations anticipate the operations studied in Chapter 6 as translations.

A discussion of unavoidable regularities found in very long strings, completes the theoretical picture.

The last section mentions the Chomsky classification of grammar types and exemplifies context-sensitive grammars, stressing the difficulty of this rarely used model.

## 2.2 Formal Language Theory

Formal language theory starts from the elementary notions of alphabet, string operations, and aggregate operations on sets of strings. By such operations complex languages can be obtained starting from simpler ones.

### 2.2.1 Alphabet and Language

An *alphabet* is a finite set of elements called *terminal symbols* or *characters*. Let  $\Sigma = \{a_1, a_2, \dots, a_k\}$  be an alphabet with  $k$  elements, i.e., its *cardinality* is  $|\Sigma| = k$ . A *string* (also called a *word*) is a sequence (i.e., an ordered set possibly with repetitions) of characters.

*Example 2.1.* Let  $\Sigma = \{a, b\}$  be the alphabet. Some strings are: *aaba*, *aaa*, *abaa*, *b*.

A *language* is a set of strings on a specified alphabet.

*Example 2.2.* For the same alphabet  $\Sigma = \{a, b\}$  three examples of languages follow:

$$\begin{aligned} L_1 &= \{aa, aaa\} \\ L_2 &= \{aba, aab\} \\ L_3 &= \{ab, ba, aabb, abab, \dots, aaabbb, \dots\} = \text{set of strings having as many} \\ &\quad \text{a's as b's} \end{aligned}$$

Notice that a formal language viewed as a set has two layers: at the first level there is an unordered set of nonelementary elements, the strings. At the second level, each string is an ordered set of atomic elements, the terminal characters.

Given a language, a string belonging to it is called a *sentence* or *phrase*. Thus *bbaa*  $\in L_3$  is a sentence of  $L_3$ , whereas *abb*  $\notin L_3$  is an *incorrect* string.

The *cardinality* or size of a language is the number of sentences it contains. For instance,  $|L_2| = |\{aba, aab\}| = 2$ . If the cardinality is finite, the language is called *finite*, too. Otherwise, there is no finite bound on the number of sentences, and the language is termed *infinite*. To illustrate,  $L_1$  and  $L_2$  are finite, but  $L_3$  is infinite.

One can observe a finite language is essentially a collection of words<sup>2</sup> sometimes called a *vocabulary*. A special finite language is the *empty* set or *language*  $\emptyset$ , which contains no sentence,  $|\emptyset| = 0$ . Usually, when a language contains just one element, the set braces are omitted writing e.g., *abb* instead of  $\{abb\}$ .

---

<sup>2</sup> In mathematical writings the terms *word* and *string* are synonymous, in linguistics a word is a string having a meaning.

It is convenient to introduce the notation  $|x|_b$  for the number of characters  $b$  present in a string  $x$ . For instance:

$$|aab|_a = 2, \quad |aba|_a = 2, \quad |baa|_c = 0$$

The *length*  $|x|$  of a string  $x$  is the number of characters it contains, e.g.:  $|ab| = 2$ ;  $|abaa| = 4$ .

Two strings

$$x = a_1a_2 \dots a_h, \quad y = b_1b_2 \dots b_k$$

are *equal* if  $h = k$  and  $a_i = b_i$ , for every  $i = 1, \dots, h$ . In words, examining the strings from left to right their respective characters coincide. Thus we obtain:

$$aba \neq baa, \quad baa \neq ba$$

## String Operations

In order to manipulate strings it is convenient to introduce several operations. For strings

$$x = a_1a_2 \dots a_h \quad y = b_1b_2 \dots b_k$$

*concatenation*<sup>3</sup> is defined as

$$x.y = a_1a_2 \dots a_hb_1b_2 \dots b_k$$

The dot may be dropped, writing  $xy$  in place of  $x.y$ . This essential operation for formal languages plays the role addition has in number theory.

*Example 2.3.* For strings

$$x = \text{well}, \quad y = \text{in}, \quad z = \text{formed}$$

we obtain

$$xy = \text{wellin}, \quad yx = \text{inwell} \neq xy$$

$$(xy)z = \text{wellin.formed} = x(yz) = \text{well.informed} = \text{wellinformed}$$

Concatenation is clearly non-commutative, that is, the identity  $xy \neq yx$  does not hold in general. The *associative* property holds:

$$(xy)z = x(yz)$$

This permits to write without parentheses the concatenation of three or more strings. The length of the result is the sum of the lengths of the concatenated strings:

$$|xy| = |x| + |y| \tag{2.1}$$

---

<sup>3</sup> Also termed *product* in mathematical works.

## Empty string

It is useful to introduce the concept of *empty* (or null) *string*, denoted by Greek epsilon  $\varepsilon$ , as the only string satisfying the identity

$$x\varepsilon = \varepsilon x = x$$

for every string  $x$ . From equality 2.1 it follows the empty string has length zero:

$$|\varepsilon| = 0$$

From an algebraic perspective, the empty string is the neutral element with respect to concatenation, because any string is unaffected by concatenating  $\varepsilon$  to the left or right.

The empty string should not be confused with the empty set: in fact  $\emptyset$  as a language contains no string, whereas the set  $\{\varepsilon\}$  contains one, the empty string.

## Substrings

Let  $x = uv$  be the concatenation of some, possibly empty, strings  $u, v$ . Then  $y$  is a *substring* of  $x$ ; moreover,  $u$  is a *prefix* of  $x$ , and  $v$  is a *suffix* of  $x$ . A substring (prefix, suffix) is called *proper* if it does not coincide with string  $x$ .

Let  $x$  be a string of length at least  $k$ ,  $|x| \geq k \geq 1$ . The notation  $Ini_k(x)$  denotes the prefix  $u$  of  $x$  having length  $k$ , to be termed the *initial* of length  $k$ .

*Example 2.4.* The string  $x = aabacba$  contains the following components:

prefixes:  $a, aa, aab, aaba, aabac, aabacb, aabacba$   
 suffixes:  $a, ba, cba, acba, bacba, abacba, aabacba$   
 substrings: all prefixes and suffixes and the internal  
 strings such as  $a, ab, ba, bacb, \dots$

Notice that  $bc$  is not a substring of  $x$ , although both  $b$  and  $c$  occur in  $x$ . The initial of length two is  $Ini_2(aabacba) = aa$ .

## Mirror reflection

The characters of a string are usually read from left to right, but it is sometimes requested to reverse the order. The *reflection* of a string  $x = a_1a_2\dots a_h$  is the string  $x^R = a_ha_{h-1}\dots a_1$ . For instance, it is

$$x = \text{roma} \quad x^R = \text{amor}$$

The following identities are immediate:

$$(x^R)^R = x \quad (xy)^R = y^R x^R \quad \varepsilon^R = \varepsilon$$

## Repetitions

When a string contains repetitions it is handy to have an operator denoting them. The  $m$ -th power ( $m \geq 1$ , integer) of a string  $x$  is the concatenation of  $x$  with itself  $m - 1$  times:

$$x^m = \underbrace{xx \dots x}_{m \text{ times}}$$

By stipulation the zero power of any string is defined to be the empty string. The complete definition is

$$\begin{cases} x^m = x^{m-1}x, & m > 0 \\ x^0 = \varepsilon \end{cases}$$

Examples:

$$\begin{array}{llll} x = ab & x^0 = \varepsilon & x^1 = x = ab & x^2 = (ab)^2 = abab \\ y = a^2 = aa & y^3 = a^2 a^2 a^2 = a^6 & & \\ \varepsilon^0 = \varepsilon & \varepsilon^2 = \varepsilon & & \end{array}$$

When writing formulas, the string to be repeated must be parenthesized, if longer than one. Thus to express the 2nd power of  $ab$ , i.e.,  $abab$ , one should write  $(ab)^2$ , not  $ab^2$ , which is the string  $abb$ .

Expressed differently, we assume the power operation takes *precedence* over concatenation. Similarly reflection takes precedence over concatenation: e.g.,  $ab^R$  returns  $ab$ , since  $b^R = b$ , while  $(ab)^R = ba$ .

### 2.2.2 Language Operations

It is straightforward to extend an operation, originally defined on strings, to an entire language: just apply the operation to all the sentences. By means of this general principle, previously defined string operations can be revisited, starting from those having one argument.

The reflection of a language  $L$  is the set of strings that are the reflection of a sentence:

$$L^R = \{x \mid \underbrace{x = y^R \wedge y \in L}_{\text{characteristic predicate}}\}$$

Here the strings  $x$  are specified by the property expressed in the so-called characteristic predicate.

Similarly the set of proper prefixes of a language  $L$  is

$$\text{Prefixes}(L) = \{y \mid x = yz \wedge x \in L \wedge y \neq \varepsilon \wedge z \neq \varepsilon\}$$

*Example 2.5.* Prefix-free language

In some applications the loss of one or more final characters of a language sentence is required to produce an incorrect string. The motivation is that the compiler is then able to detect inadvertent truncation of a sentence.

A language is *prefix-free* if none of the proper prefixes of sentences is in the language; i.e., if the set  $\text{Prefixes}(L)$  is disjoint from  $L$ .

Thus the language  $L_1 = \{x \mid x = a^n b^n \wedge n \geq 1\}$  is prefix-free since every prefix takes the form  $a^n b^m$ ,  $n > m \geq 0$  and does not satisfy the characteristic predicate.

On the other hand, the language  $L_2 = \{a^m b^n \mid m > n \geq 1\}$  contains  $a^3 b^2$  as well as its prefix  $a^3 b$ .

Similarly, operations on two strings can be extended to two languages, by letting the first and second argument span the respective language, for instance *concatenation* of languages  $L'$  and  $L''$  is defined as

$$L' L'' = \{xy \mid x \in L' \wedge y \in L''\}$$

From this the extension of the  $m$ -th *power* operation on a language is straightforward:

$$\begin{aligned} L^m &= L^{m-1} L, \quad m > 0 \\ L^0 &= \{\varepsilon\} \end{aligned}$$

Some special cases follow from previous definitions:

$$\begin{aligned} \emptyset^0 &= \{\varepsilon\} \\ L.\emptyset &= \emptyset.L = \emptyset \\ L.\{\varepsilon\} &= \{\varepsilon\}.L = L \end{aligned}$$

*Example 2.6.* Consider the languages

$$\begin{aligned} L_1 &= \{a^i \mid i \geq 0, \text{even}\} = \{\varepsilon, aa, aaaa, \dots\} \\ L_2 &= \{b^j a \mid j \geq 1, \text{odd}\} = \{ba, bbba, \dots\} \end{aligned}$$

We obtain

$$\begin{aligned} L_1 L_2 &= \{a^i . b^j a \mid (i \geq 0, \text{even}) \wedge (j \geq 1, \text{odd})\} \\ &= \{\varepsilon ba, a^2 ba, a^4 ba, \dots, \varepsilon b^3 a, a^2 b^3 a, a^4 b^3 a, \dots\} \end{aligned}$$

A common error when computing the power is to take  $m$  times the *same* string. The result is a different set, included in the power:



$$\{x \mid x = y^m \wedge y \in L\} \subseteq L^m, \quad m \geq 2$$

Thus for  $L = \{a, b\}$  with  $m = 2$  the left part is  $\{aa, bb\}$  and the right part is  $\{aa, ab, ba, bb\}$ .

*Example 2.7.* Strings of finite length

The power operation allows a concise definition of the strings of length not exceeding some integer  $k$ . Consider the alphabet  $\Sigma = \{a, b\}$ . For  $k = 3$  the language

$$\begin{aligned} L &= \{\varepsilon, a, b, aa, ab, ba, bb, aaa, aab, aba, abb, baa, bab, bba, bbb\} \\ &= \Sigma^0 \cup \Sigma^1 \cup \Sigma^2 \cup \Sigma^3 \end{aligned}$$

may also be defined as

$$L = \{\varepsilon, a, b\}^3$$

Notice that sentences shorter than  $k$  are obtained using the empty string of the base language.

Slightly changing the example, the language  $\{x \mid 1 \leq |x| \leq 3\}$  is defined, using concatenation and power, by the formula

$$L = \{a, b\}\{\varepsilon, a, b\}^2$$

### 2.2.3 Set Operations

Since a language is a set, the classical set operations, union ( $\cup$ ), intersection ( $\cap$ ), and difference ( $\setminus$ ), apply to languages; set relations, inclusion ( $\subseteq$ ), strict inclusion ( $\subset$ ), and equality ( $=$ ) apply as well.

Before introducing the complement of a language, the notion of *universal language* is needed: it is defined as the set of all strings of alphabet  $\Sigma$ , of any length, including zero.

Clearly the universal language is infinite and can be viewed as the union of all the powers of the alphabet:

$$L_{\text{universal}} = \Sigma^0 \cup \Sigma \cup \Sigma^2 \cup \dots$$

The *complement* of a language  $L$  of alphabet  $\Sigma$ , denoted by  $\neg L$ , is the set difference

$$\neg L = L_{\text{universal}} \setminus L$$

that is, the set of the strings of alphabet  $\Sigma$  that are not in  $L$ . When the alphabet is understood, the universal language can be expressed as the complement of the empty language:

$$L_{\text{universal}} = \neg \emptyset$$

*Example 2.8.* The complement of a finite language is always infinite, for instance the set of strings of any length except two is

$$\neg(\{a, b\}^2) = \varepsilon \cup \{a, b\} \cup \{a, b\}^3 \cup \dots$$

On the other hand, the complement of an infinite language may or may not be finite, as shown on one side by the complement of the universal language, on the other side by the complement of the set of even length strings with alphabet  $\{a\}$ :

$$L = \{a^{2n} \mid n \geq 0\} \quad \neg L = \{a^{2n+1} \mid n \geq 0\}$$

Moving to set difference, consider alphabet  $\Sigma = \{a, b, c\}$  and languages

$$L_1 = \{x \mid |x|_a = |x|_b = |x|_c \geq 0\}$$

$$L_2 = \{x \mid |x|_a = |x|_b \wedge |x|_c = 1\}$$

Then the differences are,

$$L_1 \setminus L_2 = \varepsilon \cup \{x \mid |x|_a = |x|_b = |x|_c \geq 2\}$$

which represents the set of strings having the same number, excluding 1, of occurrences of letters  $a, b, c$ ;

$$L_2 \setminus L_1 = \{x \mid |x|_a = |x|_b \neq |x|_c = 1\}$$

the set of strings having one  $c$  and the same number of occurrences of  $a, b$ , excluding 1.

### 2.2.4 Star and Cross

Most artificial and natural languages include sentences that can be lengthened at will, causing the number of sentences in the language to be unbounded. On the other hand, all the operations so far defined, with the exception of complement, do not allow to write a finite formula denoting an infinite language. In order to enable the definition of an infinite language, the next essential development extends the power operation to the limit.

The *star*<sup>4</sup> operation is defined as the union of all the powers of the base language:

$$L^* = \bigcup_{h=0 \dots \infty} L^h = L^0 \cup L^1 \cup L^2 \cup \dots = \varepsilon \cup L \cup L^2 \cup \dots$$

---

<sup>4</sup> Also known as Kleene's star and as closure by concatenation.

*Example 2.9.* For  $L = \{ab, ba\}$

$$L^* = \{\varepsilon, ab, ba, abab, abba, baab, baba, \dots\}$$

Every string of the star can be segmented into substrings which are sentences of the base language  $L$ .

Notice that starting with a finite base language,  $L$ , the “starred” language  $L^*$  is infinite.

It may happen that the starred and base language are identical as in

$$L = \{a^{2n} \mid n \geq 0\} \quad L^* = \{a^{2n} \mid n \geq 0\} \equiv L$$

An interesting special case is when the base is an alphabet  $\Sigma$ , then the star  $\Sigma^*$  contains all the strings<sup>5</sup> obtained by concatenating terminal characters. This language is the same as the previous *universal language* of alphabet  $\Sigma$ .<sup>6</sup>

It is clear that any formal language is a subset of the universal language of the same alphabet; the relation

$$L \subseteq \Sigma^*$$

is often written to say that  $L$  is a language of alphabet  $\Sigma$ .

Some useful properties of star:

$$\begin{array}{ll} L \subseteq L^* & \text{(monotonicity)} \\ \text{if } (x \in L^* \wedge y \in L^*) \text{ then } xy \in L^* & \text{(closure by concatenation)} \\ (L^*)^* = L^* & \text{(idempotence)} \\ (L^*)^R = (L^R)^* & \text{(commutativity of star and reflection)} \end{array}$$

*Example 2.10.* Idempotence

The monotonicity property affirms any language is included in its star. But for language  $L_1 = \{a^{2n} \mid n \geq 0\}$  the equality  $L_1^* = L_1$  follows from the idempotence property and the fact that  $L_1$  can be equivalently defined by the starred formula  $\{aa\}^*$ .

For the empty language and empty string we have the identities

$$\emptyset^* = \{\varepsilon\} \quad \{\varepsilon\}^* = \{\varepsilon\}$$

---

<sup>5</sup> The length of a sentence in  $\Sigma^*$  is unbounded but it may not be considered to be infinite. A specialized branch of this theory (see Perrin and Pin [41]) is devoted to so-called infinitary or omega-languages, which include also sentences of infinite length. They effectively model the situations when an eternal system can receive or produce messages of infinite length.

<sup>6</sup> Another name for it is *free monoid*. In algebra a monoid is a structure provided with an associative composition law (concatenation) and a neutral element (empty string).

*Example 2.11. Identifiers*

Many artificial languages assign a name or identifier to each entity (variable, file, document, subprogram, object, etc.). A usual naming rule prescribes that an identifier should be a string with initial character in  $\{A, B, \dots, Z\}$  and containing any number of letters or digits  $\{0, 1, \dots, 9\}$ , such as CICLO3A2. Using the alphabets

$$\Sigma_A = \{A, B, \dots, Z\}, \quad \Sigma_N = \{0, 1, \dots, 9\}$$

the language of identifiers  $I \subseteq (\Sigma_A \cup \Sigma_N)^*$  is

$$I = \Sigma_A(\Sigma_A \cup \Sigma_N)^*$$

To introduce a variance, prescribe that the length of identifiers should not exceed 5. Defining  $\Sigma = \Sigma_A \cup \Sigma_N$ , the language is

$$I_5 = \Sigma_A(\Sigma^0 \cup \Sigma^1 \cup \Sigma^2 \cup \Sigma^3 \cup \Sigma^4) = \Sigma_A(\varepsilon \cup \Sigma \cup \Sigma^2 \cup \Sigma^3 \cup \Sigma^4)$$

The formula expresses concatenation of language  $\Sigma_A$ , whose sentences are single characters, with the language constructed as the union of powers. A more elegant writing is

$$I_5 = \Sigma_A(\varepsilon \cup \Sigma)^4$$

*Cross*

A useful though dispensable operator, derived from star, is the *cross*<sup>7</sup>

$$L^+ = \bigcup_{h=1 \dots \infty} L^h = L \cup L^2 \cup \dots$$

It differs from the star because the union is taken excluding power zero. The following relations hold:

$$\begin{aligned} L^+ &\subseteq L^* \\ \varepsilon \in L^+ &\text{ if and only if } \varepsilon \in L \\ L^+ &= LL^* = L^*L \end{aligned}$$

*Example 2.12.*

$$\{ab, bb\}^+ = \{ab, b^2, ab^3, b^2ab, abab, b^4, \dots\}$$

$$\{\varepsilon, aa\}^+ = \{\varepsilon, a^2, a^4, \dots\} = \{a^{2n} \mid n \geq 0\}$$

Not surprisingly a language can usually be defined by various formulas, that differ by their use of operators.

---

<sup>7</sup> Or nonreflective closure by concatenation.

*Example 2.13.* The strings four or more characters long may be defined by

- concatenating the strings of length four with arbitrary strings:  $\Sigma^4 \Sigma^*$ ;
- or by constructing the th power of the set of nonempty strings:  $(\Sigma^+)^4$ .

### 2.2.5 Quotient

Operations like concatenation, star, or union lengthen the strings or increase the cardinality of the set of strings they operate upon. Given two languages, the (*right*) *quotient* operation shortens the sentences of the first language by cutting a suffix, which is a sentence of the second language. The (right) quotient of  $L'$  with respect to  $L''$  is defined as

$$L = L' /_R L'' = \{y \mid \exists y, z \text{ such that } yz \in L' \wedge z \in L''\}$$

*Example 2.14.* Let

$$L' = \{a^{2n}b^{2n} \mid n > 0\}, \quad L'' = \{b^{2n+1} \mid n \geq 0\}$$

The quotients are

$$L' /_R L'' = \{a^r b^s \mid (r \geq 2, \text{ even}) \wedge (1 \leq s < r, s \text{ odd})\} = \{a^2 b, a^4 b, a^4 b^3, \dots\}$$

$$L'' /_R L' = \emptyset$$

A dual operation is the *left quotient*  $L'' /_L L'$  that shortens the sentences of the first language by cutting a prefix which is a sentence of the second language.

Other operations will be introduced later, in order to transform or translate a formal language by replacing the terminal characters with other characters or strings.

## 2.3 Regular Expressions and Languages

Theoretical investigation on formal languages has invented various categories of languages, in a way reminiscent of the classification of numerical domains introduced much earlier by number theory. Such categories are characterized by mathematical and algorithmic properties.

The first family of formal languages is called *regular* (or rational) and can be defined by an astonishing number of different approaches. Regular languages have been independently discovered in disparate scientific fields: the study of input signals driving a sequential circuit<sup>8</sup> to a certain state, the

---

<sup>8</sup> A digital component incorporating a memory.

lexicon of programming languages modelled by simple grammar rules, and the simplified analysis of neural behavior. Later such approaches have been complemented by a logical definition based on a restricted form of predicates.

To introduce the family, the first definition will be algebraic, using the union, concatenation, and star operations; then the family will be defined by certain simple grammar rules; last, Chapter 3 describes the algorithm for recognizing regular languages in the form of an abstract machine or automaton<sup>9</sup>.

### 2.3.1 Definition of Regular Expression

A language of alphabet  $\Sigma = \{a_1, a_2, \dots, a_n\}$  is *regular* if it can be expressed by applying a finite number of times the operations of concatenation, union, and star, starting with the unitary languages<sup>10</sup>  $\{a_1\}, \{a_2\}, \dots, \{a_n\}$  or the empty language  $\emptyset$ .

More precisely a *regular expression* (r.e.) is a string  $r$  containing the terminal characters of alphabet  $\Sigma$  and the metasymbols<sup>11</sup>

. concatenation       $\cup$  union      \* star       $\emptyset$  empty set      (    )

in accordance with the following rules:

1.  $r = \emptyset$                       3.  $r = (s \cup t)$                       5.  $r = (s)^*$
2.  $r = a, a \in \Sigma$     4.  $r = (s.t)$  or  $r = (st)$

where  $s$  and  $t$  are r.e.

Parentheses may often be dropped by imposing the following precedence when applying operators: first star, then concatenation, and last union.

For improving expressivity, the symbols  $\varepsilon$  (empty string) and cross may be used in an r.e., since they are derived from the three basic operations by the identities  $\varepsilon = \emptyset^*$  and  $s^+ = s(s)^*$ .

It is customary to write the union cup ' $\cup$ ' symbol as a vertical slash ' $|$ ', called *alternative*.

Rules 1. to 5. compose the syntax of r.e., to be formalized later by means of a grammar (example 2.31, p. 32).

The meaning or denotation of an r.e.  $r$  is a language  $L_r$  over alphabet  $\Sigma$ , defined by the correspondence in Table 2.3.1.

<sup>9</sup> The language family can also be defined by the form of the logical predicates characterizing language sentences, as e.g., in [52].

<sup>10</sup> A unitary language contains one sentence.

<sup>11</sup> In order to prevent confusion between terminals and metasymbols, the latter should not be in the alphabet. If not, metasymbols must be suitably recoded to make them distinguishable.

**Table 2.1** Language denoted by a regular expression.

<i>expression</i> $r$	<i>language</i> $L_r$
1. $\varepsilon$	$\{\varepsilon\}$
2. $a \in \Sigma$	$\{a\}$
3. $s \cup t$ or also $s \mid t$	$L_s \cup L_t$
4. $s.t$ or also $st$	$L_s.L_t$
5. $s^*$	$L_s^*$

*Example 2.15.* Let  $\Sigma = \{1\}$ , where 1 may be viewed as a pulse or signal. The language denoted by expression

$$e = (111)^*$$

contains the sequences multiple of three

$$L_e = \{1^n \mid n \bmod 3 = 0\}$$

Notice that dropping the parentheses the language changes, due to the precedence of star over concatenation:

$$e_1 = 111^* = 11(1)^* \quad L_{e_1} = \{1^n \mid n \geq 2\}$$

*Example 2.16.* Integers

Let  $\Sigma = \{+, -, d\}$  where  $d$  denotes any decimal digit  $0, 1, \dots, 9$ . The expression

$$e = (+ \cup - \cup \varepsilon)dd^* \equiv (+ \mid - \mid \varepsilon)dd^*$$

produces the language

$$L_e = \{+, -, \varepsilon\}\{d\}\{d\}^*$$

of integers with or without a sign, such as  $+353, -5, 969, +001$ .

Actually the correspondence between r.e. and denoted language is so direct that it is customary to refer to the language  $L_e$  by the r.e.  $e$  itself.

A language is *regular* if it is denoted by a regular expression. The collection of all regular languages is called the *family REG* of *regular* languages.

Another simple family of languages is the collection of all *finite languages*, *FIN*. A language is in *FIN* if its cardinality is finite, as for instance the language of 32-bit binary numbers.

Comparing the *REG* and *FIN* families, it is easy to see that every finite language is regular,  $FIN \subseteq REG$ . In fact, a finite language is the union of finitely many strings  $x_1, x_2, \dots, x_k$ , each one being the concatenation of finitely many characters,  $x_i = a_1a_2 \dots a_{n_i}$ . The structure of the r.e. producing a finite language is then a union of  $k$  terms, made by concatenation of  $n_i$

characters. But *REG* includes nonfinite languages too, thus proving strict inclusion of the families,  $FIN \subset REG$ .

More language families will be introduced and compared with *REG* later.

### 2.3.2 Derivation and Language

We formalize the mechanism by which an r.e. produces a string of the language. Supposing for now the given r.e.  $e$  is fully parenthesized (except for atomic terms), we introduce the notion of *subexpression* (s.e.) in the next example:

$$e_0 = \left( \overbrace{\left( (a \cup (bb))^* \right)}^{e_1} \left( \overbrace{\left( (c^+) \cup \underbrace{(a \cup (bb))}_s \right)}^{e_2} \right) \right)$$

This r.e. is structured as concatenation of two parts  $e_1$  and  $e_2$ , to be called subexpressions. In general an s.e.  $f$  of an r.e.  $e$  is a well-parenthesized substring immediately occurring inside the outermost parentheses. This means no other well-parenthesized substring of  $e$  contains  $f$ . In the example, the substring labelled  $s$  is not s.e. of  $e_0$  but is s.e. of  $e_2$ .

When the r.e. is not fully parenthesized, in order to identify the subexpressions one has to insert (or to imagine) the missing parentheses, in agreement with operator precedence.

Notice that three or more terms, combined by union, need not to be pairwise parenthesized, because the operation is associative, as in:

$$(c^+ \cup a \cup (bb))$$

The same applies to three or more concatenated terms.

A union or repetition (star and cross) operator offers different choices for producing strings. By making a choice, one obtains an r.e. defining a less general language, which is included in the original one. We say an r.e. is a *choice* of another one in the following cases:

1.  $e_k, 1 \leq k \leq m$ , is a choice of the union  $(e_1 \cup \dots \cup e_k \cup \dots \cup e_m)$
2.  $e^m = \underbrace{e \dots e}_{m \text{ times}}, m \geq 1$ , is a choice of the expressions  $e^*, e^+$
3. the empty string is a choice of  $e^*$

Let  $e'$  be an r.e.; an r.e.  $e''$  can be derived from  $e'$  by substituting some choice for  $e'$ . The corresponding relation called *derivation* between two regular expressions  $e', e''$  is defined next.



**Definition 2.17.** Derivation<sup>12</sup>

We say  $e'$  *derives*  $e''$ , written  $e' \Rightarrow e''$ , if

$e''$  is a choice of  $e'$

or

$$e' = e_1 \dots e_k \dots e_m \text{ and } e'' = e_1 \dots e''_k \dots e_m$$

where  $e''_k$  is a choice of  $e_k$ ,  $1 \leq k \leq m$

A derivation can be applied two or more times in a row. We say  $e_0$  derives  $e_n$  in  $n$  steps, written

$$e_0 \xRightarrow{n} e_n$$

if

$$e_0 \Rightarrow e_1, \quad e_1 \Rightarrow e_2, \quad \dots, \quad e_{n-1} \Rightarrow e_n$$

The notation

$$e_0 \xRightarrow{+} e_n$$

states that  $e_0$  derives  $e_n$  in some  $n \geq 1$  steps. The case  $n = 0$  corresponds to the identity  $e_0 = e_n$  and says the derivation relation is reflective. We also write

$$e_0 \xRightarrow{*} e_n \text{ for } \left( e_0 \xRightarrow{+} e_n \right) \vee (e_0 = e_n)$$

*Example 2.18.* Immediate derivations:

$$a^* \cup b^+ \Rightarrow a^*, \quad a^* \cup b^+ \Rightarrow b^+, \quad (a^* \cup bb)^* \Rightarrow (a^* \cup bb)(a^* \cup bb)$$

Notice that the substrings of the r.e. considered must be chosen in order from external to internal, if one wants to produce all possible derivations. For instance, it would be unwise, starting from  $e' = (a^* \cup bb)^*$ , to choose  $(a^2 \cup bb)^*$ , because  $a^*$  is not an s.e. of  $e'$ . Although 2 is a correct choice for the star, such premature choice would rule out the derivation of a valid sentence such as  $a^2 b b a^3$ .

Multi-step derivations:

$$a^* \cup b^+ \Rightarrow a^* \Rightarrow \varepsilon \text{ that is } a^* \cup b^+ \xRightarrow{2} \varepsilon \text{ or also } a^* \cup b^+ \xRightarrow{+} \varepsilon$$

$$a^* \cup b^+ \Rightarrow b^+ \Rightarrow bbb \text{ or also } (a^* \cup b^+) \xRightarrow{+} bbb$$

Some expressions produced by derivation from an expression  $r$  contain the metasymbols union, star, and cross; some others just terminal characters or the empty string (and maybe some redundant parentheses which can be cancelled). The latter expressions compose the language denoted by the r.e.

The *language defined by a regular expression*  $r$  is

---

<sup>12</sup> Also called *implication*.

$$L_r = \{x \in \Sigma^* \mid r \xRightarrow{*} x\}$$

Two r.e. are *equivalent* if they define the same language.

The coming example shows that different orders of derivation may produce the same sentence.

*Example 2.19.* Consider the derivations

1.  $a^*(b \cup c \cup d)f^+ \Rightarrow aaa(b \cup c \cup d)f^+ \Rightarrow aaacf^+ \Rightarrow aaacf$
2.  $a^*(b \cup c \cup d)f^+ \Rightarrow a^*cf^+ \Rightarrow aaacf^+ \Rightarrow aaacf$

Compare derivations 1. and 2. In 1. the first choice takes the leftmost s.e. ( $a^*$ ), whereas in 2. another s.e. ( $b \cup c \cup d$ ) is taken. Since the two steps are independent of each other, they can be applied in any order. By a further step, we obtain r.e.  $aaacf^+$ , and the last step produces sentence  $aaacf$ . The last step, being independent from the others, could be performed before, after, or in between.

The example has shown that many different but equivalent orders of choice making, derive the same sentence.

## Ambiguity of Regular Expressions

The next example conceptually differs from the preceding one with respect to the way different derivations produce the same sentence.

*Example 2.20.* Ambiguous regular expression

The language of alphabet  $\{a, b\}$ , characterized by the presence of at least one  $a$ , is defined by

$$(a \cup b)^*a(a \cup b)^*$$

where the compulsory presence of  $a$  is evident. Now sentences containing two or more occurrences of  $a$  can be obtained by multiple derivations, which differ with respect to the character identified with the compulsory one of the r.e. For instance, sentence  $aa$  offers two possibilities:

$$(a \cup b)^*a(a \cup b)^* \Rightarrow (a \cup b)a(a \cup b)^* \Rightarrow aa(a \cup b)^* \Rightarrow aa\varepsilon = aa$$

$$(a \cup b)^*a(a \cup b)^* \Rightarrow \varepsilon a(a \cup b)^* \Rightarrow \varepsilon a(a \cup b) \Rightarrow \varepsilon aa = aa$$

This sentence (and the r.e. deriving it) is said to be ambiguous, because there are two structurally different derivations. On the other hand, sentence  $ba$  is not ambiguous, because there exists only one set of choices, corresponding to derivation

$$(a \cup b)^*a(a \cup b)^* \Rightarrow (a \cup b)a(a \cup b)^* \Rightarrow ba(a \cup b)^* \Rightarrow ba\varepsilon = ba$$

In order to formalize the idea of ambiguity, it helps to number the letters of the r.e.  $f$ , obtaining a *numbered* r.e.:

$$f' = (a_1 \cup b_2)^* a_3 (a_4 \cup b_5)^*$$

which defines a regular language of alphabet  $\{a_1, b_2, a_3, a_4, b_5\}$ .

An r.e.  $f$  is *ambiguous* if the language defined by the corresponding numbered r.e.  $f'$  contains distinct strings  $x, y$  such that they become identical when the numbers are erased<sup>13</sup>. For instance, strings  $a_1 a_3$  and  $a_3 a_4$  of language  $f'$  prove ambiguity of  $aa$ .

Ambiguous definitions are a source of trouble in many settings. They should be avoided in general, although they may have the advantage of concision over unambiguous definitions. The concept of ambiguity will be thoroughly studied for grammars.

### 2.3.3 Other Operators

When regular expressions are used in practice, it may be convenient to add to the *basic operators* (union, concatenation, star) the derived operators power and cross.

For further expressivity other derived operators may be practical:

Repetition from  $k$  to  $n > k$  times:  $[a]_k^n = a^k \cup a^{k+1} \cup \dots \cup a^n$

Option:  $[a] = (\varepsilon \cup a)$

Interval of ordered set: to represent any digit in the ordered set  $0, 1, \dots, 9$  the short notation is  $(0 \dots 9)$ . Similarly the notation  $(a \dots z)$  and  $(A \dots Z)$  stand for the set of lower (respectively upper) case letters.

Sometimes, other set operations are also used: intersection, set difference, and complement. Expressions using such operators are called *extended r.e.*, although the name is not standard, and one has to specify the allowed operators.

*Example 2.21.* Extended r.e. with intersection

This operator provides a straightforward formulation of the fact that valid strings must simultaneously obey two conditions. To illustrate, let  $\{a, b\}$  be the alphabet and assume a valid string must (1) contain substring  $bb$  and (2) have even length. The former condition is imposed by r.e.

$$(a \mid b)^* bb (a \mid b)^*$$

the latter by r.e.

$$((a \mid b)^2)^*$$

and the language by the r.e. extended with intersection

$$((a \mid b)^* bb (a \mid b)^*) \cap ((a \mid b)^2)^*$$

---

<sup>13</sup> Notice the empty string too may be ambiguous. Observe formulas  $(ab)^* \mid \varepsilon$  and  $(ab)^* \mid c^*$ .

The same language can be defined by a basic r.e., without intersection, but the formula is more complicated. It says substring  $bb$  can be surrounded by two strings of even length or by two strings of odd length:

$$((a | b)^2)^*bb((a | b)^2)^* \mid (a | b)((a | b)^2)^*bb(a | b)((a | b)^2)^*$$

Furthermore it is sometimes simpler to define the sentences of a language *ex negativo*, by stating a property they should not have.

*Example 2.22.* Extended r.e. with complement

Consider the set  $L$  of strings of alphabet  $\{a, b\}$  not containing  $aa$  as substring. The complement of the language is

$$\neg L = \{x \in (a | b)^* \mid x \text{ contains substring } aa\}$$

easily defined by r.e.  $(a | b)^*aa(a | b)^*$ , whence the extended r.e.

$$L = \neg((a | b)^*aa(a | b)^*)$$

The definition by a basic r.e.

$$(ab | b)^*(a | \varepsilon)$$

is, subjectively, less readable.

Actually it is not by coincidence that both preceding examples admit also an r.e. without intersection or complement. A theoretical result to be presented in Chapter 3 states that an r.e. extended with complement and intersection produces always a regular language, which by definition can be defined by a nonextended r.e. as well.

### 2.3.4 Closure Properties of REG Family

Let  $op$  be an operator to be applied to one or two languages, to produce another language. A language family is *closed by operator*  $op$  if the language, obtained applying  $op$  to any languages of the family, is in the same family.

*Property 2.23.* The family  $REG$  of regular languages is closed by the operators concatenation, union, and star (therefore also by derived operators such as cross).

The property descends from the very definition of r.e. and of  $REG$  (p. 19). In spite of its theoretical connotation, the property has practical relevance: two regular languages can be combined using the above operations, at no risk of losing the nice features of the class of regular languages. This will have an important practical consequence, to permit compositional design of

algorithms used to check if an input string is valid for a language. Furthermore we anticipate the *REG* family is closed by intersection, complement, and reflection too, which will be proved later.

The next statement provides an alternative definition of family *REG*.

*Property 2.24.* The family *REG* of regular languages is *the smallest* language family such that: (i) it contains all finite languages and (ii) it is closed by concatenation, union, and star.

The proof is simple. Suppose by contradiction a family exists  $F \subset REG$ , which is closed by the same operators and contains all finite languages. Consider any language  $L_e$  defined by an r.e.  $e$ ; the language is obtained by repeated applications of the operators present in  $e$ , starting with some finite languages consisting of single characters. It follows from the hypothesis that language  $L(e)$  belongs also to family  $F$ , which then contains any regular language, contradicting the strict inclusion  $F \subset REG$ .

We anticipate other families exist which are closed by the same operators of property 2.23. Chief among them is the family *CF* of context-free languages, to be introduced soon. From statement 2.24 follows a containment relation between the two families,  $REG \subset CF$ .

## 2.4 Linguistic Abstraction

If one recalls the programming languages he is familiar with, he may observe that, although superficially different in their use of keywords and separators, they are often quite similar at a deeper level. By shifting focus from concrete to abstract syntax we can reduce the bewildering variety of language constructs to a few essential structures. The verb “to abstract” means<sup>14</sup>

consider a concept without thinking of a specific example.

Abstracting away from the actual characters representing a language construct we perform a linguistic abstraction. This is a language transformation that replaces the terminal characters of the concrete language with other ones taken from an abstract alphabet. Abstract characters should be simpler and suitable to represent similar constructs from different artificial languages.<sup>15</sup>

By this approach the abstract syntax structures of existing artificial languages are easily described as composition of few elementary paradigms, by means of standard language operations: union, iteration, substitution (later defined). Starting from the abstract language, a concrete or real language is

---

<sup>14</sup> From WordNet 2.1.

<sup>15</sup> The idea of language abstraction is inspired by research in linguistics aiming at discovering the underlying similarities of human languages, disregarding morphological and syntactic differences.

obtained by the reverse transformation, metaphorically called coating with syntax sugar.

Factoring a language into its abstract and concrete syntax pays off in several ways. When studying different languages it affords much conceptual economy. When designing compilers, abstraction helps for portability across different languages, if compiler functions are designed to process abstract, instead of concrete, language constructs. Thus parts of, say, a C compiler can be reused for similar languages like FORTRAN or Pascal.

The surprisingly few abstract paradigms in use, will be presented in this chapter, starting from the ones conveniently specified by regular expressions, the lists.

### 2.4.1 Abstract and Concrete Lists

An abstract *list* contains an unbounded number of elements  $e$  of the same type. It is defined by r.e.  $e^+$  or  $e^*$ , if elements can be missing.

An element for the moment should be viewed as a terminal character; but in later refinements, the element may become a string from another formal language: think e.g., of a list of numbers.

#### Lists with Separators and Opening/Closing Marks

In many concrete cases, adjacent elements must be separated by strings called *separators*,  $s$  in abstract syntax. Thus in a list of numbers, a separator should delimit the end of a number and the beginning of the next one.

A *list with separators* is defined by r.e.  $e(se)^*$ , saying the first element can be followed by zero or more pairs  $se$ . The equivalent definition  $(es)^*e$  differs by giving evidence to the last element.

In many concrete cases there is another requirement, intended for legibility or computer processing: to make the start and end of the list easily recognizable by prefixing and suffixing some special signs: in the abstract, the initial character or *opening mark*  $i$ , and the final character or *closing mark*  $f$ .

Lists with separators and opening/closing marks are defined as

$$ie(se)^*f$$

*Example 2.25.* Some concrete lists

Lists are everywhere in languages, as shown by typical examples.

Instruction block:  $begin\ instr_1; instr_2; \dots instr_n end$

where *instr* possibly stands for assignment, go to, if-statement, write-statement, etc. Corresponding abstract and concrete terms are:

<i>abstract alphabet</i>	<i>concrete alphabet</i>
$i$	$begin$
$e$	$instr$
$s$	$;$
$f$	$end$

Procedure parameters: as in

$$\underbrace{\text{procedure } WRITE}_{i}(\underbrace{par_1}_e, \underbrace{par_2, \dots, par_n}_s)$$

Should an empty parameter list be legal, as e.g.,  $\text{procedure } WRITE()$ , the r.e. becomes  $i[e(se)^*]f$ .

Array definition:  $\underbrace{\text{array } MATRIX}_{i}[\underbrace{'int_1'}_e, \underbrace{int_2, \dots, int_n}_s] \underbrace{' }_f$

where each  $int$  is an interval such as 10...50.

## Substitution

The above examples illustrate the mapping from concrete to abstract symbols. Language designers find it useful to work by stepwise refinement, as done in any branch of engineering, when a complex system is divided into its components, atomic or otherwise. To this end, we introduce the new language operation of substitution, that replaces a terminal character of a language termed the *source*, with a sentence of another language called the *target*. As always  $\Sigma$  is the source alphabet and  $L \subseteq \Sigma^*$  the source language. Consider a sentence of  $L$  containing one or more occurrences of a source character  $b$ :

$$x = a_1 a_2 \dots a_n \quad \text{where for some } i, a_i = b$$

Let  $\Delta$  be another alphabet, called target, and  $L_b \subseteq \Delta^*$  be the *image language* of  $b$ . The *substitution* of language  $L_b$  for  $b$  in string  $x$  produces a set of strings, that is, a language of alphabet  $(\Sigma \setminus \{b\}) \cup \Delta$ , defined as

$$\{y \mid y = y_1 y_2 \dots y_n \wedge (\text{if } a_i \neq b \text{ then } y_i = a_i \text{ else } y_i \in L_b)\}$$

Notice all characters other than  $b$  do not change. By the usual approach the substitution can be defined on the whole source language, by applying the operation to every source sentence.

*Example 2.26.* Example 2.25 continued

Resuming the case of a parameter list, the abstract syntax is

$$ie(se)^*f$$

and the substitutions to be applied are tabulated below:

<i>abstract char.</i>	<i>imagine</i>
<i>i</i>	$L_i = \text{procedure } \langle \text{procedure identifier} \rangle ($
<i>e</i>	$L_e = \langle \text{parameter identifier} \rangle$
<i>s</i>	$L_s = ,$
<i>f</i>	$L_f = )$

For instance, the opening mark *i* is replaced with a string of language  $L_i$ , where the procedure identifier has to agree with the rules of the technical language.

Clearly the target languages of the substitution depend on the syntax sugar of the concrete language intended for.

Notice the four substitutions are independent and can be applied in any order.

*Example 2.27.* Identifiers with underscore

In certain programming languages, long mnemonic identifier names can be constructed by appending alphanumeric strings separated by a low dash: thus *LOOP3\_OF\_35* is a legal identifier. More precisely the first string must initiate with a letter, the others may contain letters and digits, and adjacent dashes are forbidden, as well as a trailing dash.

At first glance the language is a nonempty list of strings *s*, separated by a dash:

$$s(\_s)^*$$

However, the first string should be different from the others and may be taken to be the opening mark of a possibly empty list:

$$i(\_s)^*$$

Substituting to *i* the language  $(A \dots Z)(A \dots Z \mid 0 \dots 9)^*$ , and to *s* the language  $(A \dots Z \mid 0 \dots 9)^+$ , the final r.e. is obtained.

This is an overly simple instance of syntax design by abstraction and stepwise refinement, a method to be further developed now and after the introduction of grammars.

Other language transformations are studied in Chapter 6.

## Hierarchical or Precedence Lists

A recurrent construct is a list such that each element is in turn a list of a different type. The first list is attached to level 1, the second to level 2, and so on. The present abstract structure, called hierarchical list, is restricted to lists with a bounded number of levels. The case when levels are unbounded is studied later using grammars, under the name of nested structures.

A hierarchical list is also called a *list with precedences*, because a list at level *k* bounds its elements more strongly than the list at level *k* − 1; in other



words the elements at higher level must be assembled into a list, and each becomes an element at next lower level.

Each level may have opening/closing marks and separator; such delimiters are usually distinct level by level, in order to avoid confusion.

The structure of a  $k \geq 2$  levels hierarchical list is

$$\begin{aligned} list_1 &= i_1 list_2 (s_1 list_2)^* f_1 \\ list_2 &= i_2 list_3 (s_2 list_3)^* f_2 \\ &\dots \\ list_k &= i_k e_k (s_k e_k)^* f_k \end{aligned}$$

Notice the last level alone may contain atomic elements. But a common variant permits at any level  $k$  atomic elements  $e_k$  to occur side by side with lists of level  $k + 1$ . Some concrete examples follow.

*Example 2.28.* Two hierarchical lists

Block of print instructions: *begin instr*<sub>1</sub>; *instr*<sub>2</sub>; ... *instr*<sub>*n*</sub> *end*

where *instr* is a print instruction, *WRITE*(*var*<sub>1</sub>, *var*<sub>2</sub>, ..., *var*<sub>*n*</sub>), i.e., a list (from example 2.25). There are two levels:

Level 1: list of instructions *instr*, opened by *begin*, separated by semicolon and closed by *end*.

Level 2: list of variables *var* separated by comma, with  $i_2 = \textit{WRITE}(\text{ and } f_2 = \text{ })$ .

Arithmetic expression not using parentheses: the precedence levels of operators determine how many levels there are in the list. For instance, the operators  $\times, \div$  and  $+, -$  are layered on two levels and the string

$$3 + \underbrace{5 \times 7 \times 4}_{\text{term}_1} - \underbrace{8 \times 2 \div 5}_{\text{term}_2} + 8 + 3$$

is a two-level list, with neither opening nor closing mark. At level one we find a list of terms ( $e_1 = \text{term}$ ) separated by the signs  $+$  and  $-$ , i.e., by lower precedence operators. At level two we see a list of numbers, separated by higher precedence signs  $\times, \div$ .

One may go further and introduce a third level having the exponentiation sign “\*\*” as separator.

Hierarchical structures are of course omnipresent in natural languages as well. Think of a list of nouns

father, mother, son, and daughter

Here we may observe a difference with respect to the abstract paradigm: the penultimate element has a distinct separator, possibly in order to warn the listener of an utterance that the list is approaching the end. Furthermore,

items in the list may be enriched by second-level qualifiers, such as a list of adjectives.

In all sorts of documents and written media, hierarchical lists are extremely common. For instance, a book is a list of chapters, separated by white pages, between a front and back cover. A chapter is a list of sections; a section a list of paragraphs, and so on.

## 2.5 Context-Free Generative Grammars

We start the study of the context-free language family, which plays the central role in compilation. Initially invented by linguists for natural languages in the 1950s, context-free grammars have proved themselves extremely useful in computer science applications: all existing technical languages have been defined using such grammars. Moreover, since the early 1960s efficient algorithms have been found to analyze, recognize, and translate sentences of context-free languages. This chapter presents the relevant properties of context-free languages, illustrates their application by many typical cases, and lastly positions this formal model in the classical hierarchy of grammars and computational models due to Chomsky.

### 2.5.1 *Limits of Regular Languages*

Regular expressions, though quite practical for describing list and related paradigms, falls short of the capacity needed to define other frequently occurring constructs. A case are the block structures (or nested parentheses) offered by many technical languages, schematized by

*begin begin**begin . . . end**begin . . . end . . . end end*

*Example 2.29.* Simple block structure

In brief, let  $\{b, e\}$  be the alphabet, and consider a somewhat limited case of nested structures, such that all opening marks precede all closing marks. Clearly, opening/closing marks must have identical count:

$$L_1 = \{b^n e^n \mid n \geq 1\}$$

We argue this language cannot be defined by a regular expression, deferring the formal proof to a later section. In fact, since strings must have all  $b$ 's left of any  $e$ , either we write an overly general r.e. such as  $b^+e^+$ , which accepts illegal strings like  $b^3e^5$ ; or we write a too restricted r.e. that exhaustively lists a finite sample of strings up to a bounded length. On the other hand, if we

comply with the condition that the count of the two letters is the same by writing  $(be)^+$ , illegal strings like *bebe* creep in.

For defining this and other languages, regular or not, we move to the formal model of *generative* grammars.

## 2.5.2 Introduction to Context-Free Grammars

A generative *grammar* or *syntax*<sup>16</sup> is a set of simple rules that can be repeatedly applied in order to generate all and only the valid strings.

*Example 2.30.* Palindromes

The language to be defined

$$L = \{uu^R \mid u \in \{a, b\}^*\} = \{\varepsilon, aa, bb, abba, baab, \dots, abbbba, \dots\}$$

contains even-length strings having specular symmetry, called palindromes. The following grammar  $G$  contains three rules:

$$pal \rightarrow \varepsilon \qquad pal \rightarrow a \, pal \, a \qquad pal \rightarrow b \, pal \, b$$

The arrow ‘ $\rightarrow$ ’ is a metasymbol, exclusively used to separate the left from the right part of a rule.

To derive the strings, just replace symbol ‘pal’, termed *nonterminal*, with the right part of a rule, for instance:

$$pal \Rightarrow a \, pal \, a \Rightarrow ab \, pal \, ba \Rightarrow abb \, pal \, bba \Rightarrow \dots$$

The derivation process can be chained and terminates when the last string obtained no longer contains a nonterminal symbol; at that moment the generation of the sentence is concluded. We complete the derivation:

$$abb \, pal \, bba \Rightarrow abb\varepsilon bba = abbbba$$

(Incidentally the language of palindromes is not regular.)

Next we enrich the example into a list of palindromes separated by commas, exemplified by sentence *abba, bbaabb, aa*. The grammar adds two list-generating rules to the previous ones:

$$\begin{array}{ll} list \rightarrow pal, list & pal \rightarrow \varepsilon \\ list \rightarrow pal & pal \rightarrow a \, pal \, a \\ & pal \rightarrow b \, pal \, b \end{array}$$

---

<sup>16</sup> Sometimes the term *grammar* has a broader connotation than *syntax*, as when rules for computing the meaning of sentences are added to rules for enumerating them. When necessary, the intended meaning of the term will be made clear.

The first rule says: the concatenation of palindrome, comma, and list produces a (longer) list. The second says a list can be made of one palindrome.

Now there are two nonterminal symbols: *list* and *pal*; the former is termed *axiom* because it defines the designated language, the latter defines certain component substrings, also called *constituents* of the language, the palindromes.

*Example 2.31.* Metalanguage of regular expressions

A regular expression defining a language over a fixed terminal alphabet, say  $\Sigma = \{a, b\}$ , is a formula, that is, a string over the alphabet  $\Sigma_{r.e.} = \{a, b, \cup, *, \emptyset, (, )\}$ ; such strings can be viewed in turn as sentences of a language.

Following the definition of r.e. on p. 18, this language is generated by the following syntax  $G_{r.e.}$ :

- |                                 |  |
|---------------------------------|--|
| 1. $expr \rightarrow \emptyset$ | 4. $expr \rightarrow (expr \cup expr)$ |
| 2. $expr \rightarrow a$         | 5. $expr \rightarrow (expr \, expr)$   |
| 3. $expr \rightarrow b$         | 6. $expr \rightarrow (expr)^*$         |

where numbering is only for reference. A derivation is

$$\begin{aligned}
 expr &\Rightarrow_4 (expr \cup expr) \Rightarrow_5 ((expr \, expr) \cup expr) \Rightarrow_2 ((a \, expr) \cup expr) \Rightarrow_6 \\
 &\Rightarrow ((a(expr)^*) \cup expr) \Rightarrow_4 ((a((expr \cup expr))^*) \cup expr) \Rightarrow_2 \\
 &\Rightarrow ((a((a \cup expr))^*) \cup expr) \Rightarrow_3 ((a((a \cup b))^*) \cup expr) \Rightarrow_3 ((a((a \cup b))^*) \cup b) = e
 \end{aligned}$$

Since the generated string can be interpreted as an r.e., it defines a second language of alphabet  $\Sigma$ :

$$L_e = \{a, b, aa, ab, aaa, aba, \dots\}$$

the set of strings starting with letter  $a$ , plus string  $b$ .

A word of caution: this example displays two levels of languages, since the syntax defines certain strings to be understood as definitions of other languages. To avoid terminological confusion, we say the syntax stays at the *metalinguistic* level, that is, over the linguistic level; or that the syntax is a *metagrammar*.

To set the two levels apart, it helps to consider the alphabets: at meta-level the alphabet is  $\Sigma_{r.e.} = \{a, b, \cup, *, \emptyset, (, )\}$ , whereas the final language has alphabet  $\Sigma = \{a, b\}$ , devoid of metasymbols.

An analogy with human language may also clarify the issue. A grammar of Russian can be written in, say, English. Then it contains both Cyrillic and Latin characters. Here English is the metalanguage and Russian the final language, which only contains Cyrillic characters.

Another illustration of language versus metalanguage is provided by XML, the metanotation used to define a variety of Web document types.

**Definition 2.32.** A *context-free (CF)* (or type 2 or BNF<sup>17</sup>) grammar  $G$  is defined by four entities:

1.  $V$ , *nonterminal alphabet*, a set of symbols termed *nonterminals* (or meta-symbols).
2.  $\Sigma$ , *terminal alphabet*.
3.  $P$ , a set of syntactic *rules* (or *productions*).
4.  $S \in V$ , a particular nonterminal termed *axiom*.

A rule of  $P$  is an ordered pair  $X \rightarrow \alpha$ , with  $X \in V$  and  $\alpha \in (V \cup \Sigma)^*$ . Two or more rules

$$X \rightarrow \alpha_1, X \rightarrow \alpha_2, \dots, X \rightarrow \alpha_n$$

with the same left part  $X$  can be concisely grouped in

$$X \rightarrow \alpha_1 \mid \alpha_2 \mid \dots \mid \alpha_n \quad \text{or} \quad X \rightarrow \alpha_1 \cup \alpha_2 \cup \dots \cup \alpha_n$$

We say  $\alpha_1, \alpha_2, \dots, \alpha_n$  are the *alternatives* of  $X$ .

### 2.5.3 Conventional Grammar Representations

To prevent confusion, the metasymbols ' $\rightarrow$ ', ' $\mid$ ', ' $\cup$ ', ' $\varepsilon$ ' should not be used for terminal and nonterminal symbols; moreover, the terminal and nonterminal alphabets should be disjoint. In professional and scientific practice a few different styles are used to represent terminals and nonterminals, as specified in Table 2.2. The grammar of example 2.30 in the first style becomes:

$$\begin{aligned} < \text{sentence} > \rightarrow \varepsilon \\ < \text{sentence} > \rightarrow a < \text{sentence} > a \\ < \text{sentence} > \rightarrow b < \text{sentence} > b \end{aligned}$$

Alternative rules may be grouped together:

$$< \text{sentence} > \rightarrow \varepsilon \mid a < \text{sentence} > a \mid b < \text{sentence} > b$$

If a technical grammar is large, of the order of some hundred rules, it should be written with care in order to facilitate searching for specific definitions, making changes, and cross referencing. Nonterminals should be identified by self-explanatory names and rules should be divided into sections and numbered for reference.

On the other hand, in very simple examples, the third style is more suitable,

---

<sup>17</sup> Type 2 comes from Chomsky's classification. Backus Normal Form, or also Backus Naur Form, comes from the names of John Backus and Peter Naur, who pioneered the use of such grammars for programming language definition.

**Table 2.2** Different styles for writing grammars.

<i>Nonterminals</i>	<i>Terminals</i>	<i>Examples</i>
words between angle brackets, for instance: $\langle sentence \rangle$ , $\langle list\ of\ sentences \rangle$	written as they are, without special marks	$\langle if\ sentence \rangle \rightarrow$ $if\ \langle cond \rangle\ then\ \langle sentence \rangle$ $else\ \langle sentence \rangle$
words written as they are, without special marks; may not contain blank spaces, for instance: $sentence$ , $list\_of\_sentences$	written in black, in italic or quoted, for instance:  $a\ then$  $'a'\ 'then'$	$if\_sentence \rightarrow$ $if\ cond\ then\ sentence\ else\ sentence$ or $if\_sentence \rightarrow$ $'if'\ cond\ 'then'\ sentence\ 'else'\ sentence$
uppercase Latin letters; terminal and nonterminal alphabets disjoint	written as they are, without special marks	$F \rightarrow if\ C\ then\ D\ else\ D$

i.e., to have disjoint short symbols for terminals and nonterminals. In this book, we often adopt for simplicity the following style:

- lowercase Latin letters near the beginning of the alphabet  $\{a, b, \dots\}$  for terminal characters;
- uppercase Latin letters  $\{A, B, \dots, Z\}$  for nonterminal symbols;
- lowercase Latin letters near the end of the alphabet  $\{r, s, \dots, z\}$  for strings over  $\Sigma^*$  (i.e. including only terminals);
- lowercase Greek letters  $\{\alpha, \beta, \dots\}$  for strings over the combined alphabets  $(V \cup \Sigma)^*$ .

**Types of Rules**

In grammar studies rules may be classified depending on their form, with the aim of making the study of language properties more immediate. For future reference we list in Table 2.3 some common types of rules along with their technical names. Each rule type is next schematized, with symbols adhering to the following stipulations:  $a, b$  are terminals,  $u, v, w$  denote possibly empty strings of terminals,  $A, B, C$  are nonterminals,  $\alpha, \beta$  denote possibly empty strings containing terminals and nonterminals; lastly  $\sigma$  denotes a string of nonterminals.

The classification is based on the form of the right part RP of a rule, excepting the recursive classes that also consider the left part LP. We omit any part of a rule that is, irrelevant for the classification. Left- and right-

**Table 2.3** Classification of grammar rules.

<i>Class and Description</i>	<i>Examples</i>
<i>terminal</i> : RP contains terminals or the empty string	$\rightarrow u \mid \epsilon$
<i>empty (or null)</i> : RP is empty	$\rightarrow \epsilon$
<i>initial</i> : LP is the axiom	$S \rightarrow$
<i>recursive</i> : LP occurs in RP	$A \rightarrow \alpha A \beta$
<i>left-recursive</i> : LP is prefix of RP	$A \rightarrow A \beta$
<i>right-recursive</i> : LP is suffix of RP	$A \rightarrow \beta A$
<i>left and right-recursive</i> : conjunction of two previous cases	$A \rightarrow A \beta A$
<i>copy or categorization</i> : RP is a single nonterminal	$A \rightarrow B$
<i>linear</i> : at most one nonterminal in RP	$\rightarrow u B v \mid w$
<i>right-linear</i> (type 3): as linear but nonterminal is suffix	$\rightarrow u B \mid w$
<i>left-linear</i> (type 3): as linear but nonterminal is prefix	$\rightarrow B v \mid w$
<i>homogeneous normal</i> : $n$ nonterminals or just one terminal	$\rightarrow A_1 \dots A_n \mid a$
<i>Chomsky normal</i> (or homogeneous of degree 2): two nonterminals or just one terminal	$\rightarrow B C \mid a$
<i>Greibach normal</i> : one terminal possibly followed by nonterminals	$\rightarrow a \sigma \mid b$
<i>operator normal</i> : two nonterminals separated by a terminal (operator); more generally, strings devoid of adjacent nonterminals	$\rightarrow A a B$

linear forms are also known as *type 3* grammars from Chomsky classification. Most rule types will occur in the book; the remaining ones are listed for general reference.

We shall see that some of the grammar forms can be forced on any given grammar, leaving the language unchanged. Such forms are called *normal*.

### 2.5.4 Derivation and Language Generation

We reconsider and formalize the notion of string derivation. Let  $\beta = \delta A \eta$  be a string containing a nonterminal, where  $\delta$  and  $\eta$  are any, possibly empty strings. Let  $A \rightarrow \alpha$  be a rule of  $G$  and let  $\gamma = \delta \alpha \eta$  be the string obtained replacing in  $\beta$  nonterminal  $A$  with the right part  $\alpha$ .

The relation between such two strings is called *derivation*. We say that  $\beta$  *derives*  $\gamma$  for grammar  $G$ , written

$$\beta \xRightarrow[G]{} \gamma$$

or more simply  $\beta \Rightarrow \gamma$  when the grammar name is understood. Rule  $A \rightarrow \alpha$  is applied in such derivation and string  $\alpha$  *reduces* to nonterminal  $A$ .

Consider now a chain of derivations of length  $n \geq 0$ :

$$\beta_0 \Rightarrow \beta_1 \Rightarrow \dots \Rightarrow \beta_n$$

shortened to

$$\beta_0 \xRightarrow{n} \beta_n$$

If  $n = 0$ , for every string  $\beta$  we posit  $\beta \xRightarrow{0} \beta$ , that is, the derivation relation is reflexive.

To express derivations of any length we write

$$\beta_0 \xRightarrow{*} \beta_n \text{ (resp. } \beta_0 \xRightarrow{\pm} \beta_n)$$

if the length of the chain is  $n \geq 0$  (resp.  $n \geq 1$ ).

The *language generated* or defined by a grammar  $G$  starting from nonterminal  $A$  is

$$L_A(G) = \{x \in \Sigma^* \mid A \xRightarrow{\pm} x\}$$

It is the set of terminal strings deriving in one or more steps from  $A$ .

If the nonterminal is the axiom  $S$ , we have the *language generated by  $G$* :

$$L(G) = L_S(G) = \{x \in \Sigma^* \mid S \xRightarrow{\pm} x\}$$

In some cases we need to consider derivations producing strings still containing nonterminals. A *string form* generated by  $G$  starting from nonterminal  $A \in V$ , is a string  $\alpha \in (V \cup \Sigma)^*$  such that  $A \xRightarrow{*} \alpha$ . In particular, if  $A$  is the axiom, the string is termed *sentential form*. Clearly a sentence is a sentential form devoid of nonterminals.

*Example 2.33.* Book structure

The grammar defines the structure of a book, containing a front page ( $f$ ) and a nonempty series (derived from nonterminal  $A$ ) of chapters; each one starts with a title ( $t$ ) and contains a nonempty series (derived from  $B$ ) of lines ( $l$ ). Grammar  $G_l$ :

$$\begin{array}{l} S \rightarrow fA \\ A \rightarrow AtB \mid tB \\ B \rightarrow lB \mid l \end{array}$$

Some derivations are listed. From  $A$  the string form  $tBtB$  and the string  $tlttl \in L_A(G_l)$ ; from the axiom  $S$  sentential forms  $fAtlB$ ,  $ftBtB$  and sentence  $ftltll$ .

The language generated from  $B$  is  $L_B(G_l) = l^+$ ; the language  $L(G_l)$  generated by  $G_l$  is defined by the r.e.  $f(tl^+)^+$ , showing the language is in the *REG* family. In fact, this language is a case of an abstract hierarchical list.

A language is *context-free* if a context-free grammar exists that generates it. The family of context free languages is denoted by *CF*.

Two grammars  $G$  and  $G'$  are *equivalent* if they generate the same language, i.e.,  $L(G) = L(G')$ .

*Example 2.34.* The next grammar  $G_{l2}$  is clearly equivalent to  $G_l$  of example 2.33:



$$\begin{array}{l}
S \rightarrow fX \\
X \rightarrow XtY \mid tY \\
Y \rightarrow lY \mid l
\end{array}$$

since the only change affects the way nonterminals are identified. Also the following grammar  $G_{l3}$

$$\begin{array}{l}
S \rightarrow fA \\
A \rightarrow AtB \mid tB \\
B \rightarrow Bl \mid l
\end{array}$$

is equivalent to  $G_l$ . The only difference is in row three, which defines  $B$  by a left-recursive rule, instead of the right-recursive rule used in  $G_l$ . Clearly any derivations of length  $n \geq 1$

$$B \xrightarrow[n]{G_l} l^n \quad \text{and} \quad B \xrightarrow[n]{G_{l3}} l^n$$

generate the same language  $L_B = l^+$ .

### 2.5.5 Erroneous Grammars and Useless Rules

When writing a grammar attention should be paid that all nonterminals are defined and that each one effectively contributes to the production of some sentence. In fact, some rules may turn out to be unproductive.

A grammar  $G$  is called *clean* (or *reduced*) under the following conditions:

1. every nonterminal  $A$  is *reachable* from the axiom, i.e., there exists derivation  $S \xRightarrow{*} \alpha A \beta$ ;
2. every nonterminal  $A$  is *well-defined*, i.e., it generates a nonempty language,  $L_A(G) \neq \emptyset$ .

It is often straightforward to check by inspection whether a grammar is clean. The following algorithm formalizes the checks.

#### Grammar Cleaning

The algorithm operates in two phases, first pinpointing the nondefined nonterminals, then the unreachable ones. Lastly the rules containing nonterminals of either type can be cancelled.

Phase 1. Compute the set  $DEF \subseteq V$  of well-defined nonterminals.

The set  $DEF$  is initialized with the nonterminals of terminal rules, those having a terminal string as right part:

$$DEF := \{A \mid (A \rightarrow u) \in P, \text{ with } u \in \Sigma^*\}$$

Then the next transformation is applied until convergence is reached:

$$DEF := DEF \cup \{B \mid (B \rightarrow D_1 D_2 \dots D_n) \in P\}$$

where every  $D_i$  is a terminal or a nonterminal symbol present in  $DEF$ .

At each iteration two outcomes are possible:

- a new nonterminal is found having as right part a string of symbols that are well-defined nonterminals or terminals, or else
- the termination condition is reached.

The nonterminals belonging to the complement set  $V \setminus DEF$  are nondefined and should be eliminated.

Phase 2. A nonterminal is reachable from the axiom, if, and only if, there exists a path in the following graph, which represents a relation between nonterminals, called *produce*:

$$A \xrightarrow{\text{produce}} B$$

saying that  $A$  *produces*  $B$  if, and only if, there exists a rule  $A \rightarrow \alpha B \beta$ , where  $A, B$  are nonterminals and  $\alpha, \beta$  are any strings.

Clearly  $C$  is reachable from  $S$  if, and only if, in this graph there exists an oriented path from  $S$  to  $C$ . The unreachable nonterminals are the complement with respect to  $V$ . They should be eliminated because they do not contribute to the generation of any sentence.

Quite often the following requirement is added to the above cleanness conditions.

- $G$  should not permit *circular derivations*  $A \stackrel{+}{\Rightarrow} A$ .

The reason is such derivations are inessential, because, if string  $x$  is obtained by means of a circular derivation  $A \Rightarrow A \Rightarrow x$ , it can also be obtained by the shorter derivation  $A \Rightarrow x$ .

Moreover, circular derivations cause ambiguity (a negative phenomenon later discussed).

In this book we assume grammars are always clean and noncircular.

*Example 2.35.* Unclean examples

- The grammar with rules  $\{S \rightarrow aASb, A \rightarrow b\}$  generates nothing.
- The grammar  $G$  with rules  $\{S \rightarrow a, A \rightarrow b\}$  has an unreachable nonterminal  $A$ ; the same language  $L(G)$  is generated by the clean grammar  $\{S \rightarrow a\}$ .
- Circular derivation:  
The grammar with rules  $\{S \rightarrow aASb \mid A, A \rightarrow S \mid c\}$  presents the circular derivation  $S \Rightarrow A \Rightarrow S$ . The grammar  $\{S \rightarrow aSSb \mid c\}$  is equivalent.

- Notice that circularity may also come from the presence of an empty rule, as for instance in the following grammar fragment:

$$X \rightarrow XY \mid \dots \quad Y \rightarrow \varepsilon \mid \dots$$

Finally we observe a grammar, although clean, may still contain redundant rules, as the next one:

*Example 2.36.* Double rules

$$\begin{array}{ll} 1. S \rightarrow aASb & 4. A \rightarrow c \\ 2. S \rightarrow aBSb & 5. B \rightarrow c \\ 3. S \rightarrow \varepsilon & \end{array}$$

One of the pairs (1,4) and (2,5), which generate exactly the same sentences, should be deleted.

### 2.5.6 Recursion and Language Infinity

An essential property of most technical languages is to be infinite. We study how this property follows from the form of grammar rules. In order to generate an unbounded number of strings, the grammar must be able to derive strings of unbounded length. To this end, recursive rules are necessary, as next argued.

An  $n \geq 1$  steps derivation  $A \xRightarrow{n} xAy$  is called *recursive* (*immediately* recursive if  $n = 1$ ); similarly nonterminal  $A$  is called recursive. If  $x$  (resp.  $y$ ) is empty, the recursion is termed *left* (resp. *right*).

*Property 2.37.* Let  $G$  be a grammar clean and devoid of circular derivations. The language  $L(G)$  is infinite if, and only if,  $G$  has a recursive derivation.

*Proof.* Clearly without recursive derivations, any derivation has bounded length, therefore any sentence too is bounded in length, and  $L(G)$  would be finite.

Conversely, assume  $G$  offers a recursive derivation  $A \xRightarrow{n} xAy$ , with not both  $x$  and  $y$  empty by the noncircularity hypothesis. Then the derivation  $A \xRightarrow{+} x^m Ay^m$  exists, for every  $m \geq 1$ . Since  $G$  is clean,  $A$  can be reached from the axiom by a derivation  $S \xRightarrow{*} uAv$ , and also  $A$  derives at least one terminal string  $A \xRightarrow{+} w$ . Combining the derivations, we obtain

$$S \xRightarrow{*} uAv \xRightarrow{+} ux^m Ay^m v \xRightarrow{+} ux^m wy^m v, \quad (m \geq 1)$$

that generates an infinite language.

In order to see whether a grammar has recursions, we examine the binary relation *produce* of p. 38: a grammar does not have a recursion if, and only if, the graph of the relation has no circuit.

We illustrate by two grammars generating a finite and infinite language.

*Example 2.38.* Finite language

$$S \rightarrow aBc \quad B \rightarrow ab \mid Ca \quad C \rightarrow c$$

The grammar does not have a recursion and allows just two derivations, defining the finite language  $\{aabc, acac\}$ .

The next example is a most common paradigm of so many artificial languages. It will be replicated and transformed over and over in the book.

*Example 2.39.* Arithmetic expressions

The grammar

$$G = (\{E, T, F\}, \{i, +, *, \cdot, (, ), P, E\})$$

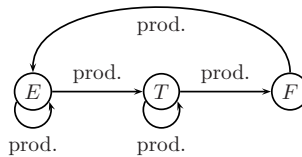
contains the rules

$$E \rightarrow E + T \mid T \quad T \rightarrow T * F \mid F \quad F \rightarrow (E) \mid i$$

The language

$$L(G) = \{i, i + i + i, i * i, (i + i) * i, \dots\}$$

is the set of arithmetic expressions over the letter  $i$ , with signs of sum and product and parentheses. Nonterminal  $F$  (factor) is nonimmediately recursive;  $T$  (term) and  $E$  (expression) are immediately recursive, both to the left. Such properties are evident from the circuits in the graph of the *produce* relation:



Since the grammar is clean and noncircular, the language is infinite.

### 2.5.7 Syntax Trees and Canonical Derivations

The process of derivation can be visualized as a syntax tree for better legibility. A *tree* is an oriented and ordered graph not containing a circuit, such that every pair of nodes is connected by exactly one oriented path. An arc  $\langle N_1 \rightarrow N_2 \rangle$  defines the  $\langle \text{father}, \text{son} \rangle$  relation, customarily visualized from top to bottom as in genealogical trees. The siblings of a node are ordered from

left to right. The *degree* of a node is the number of its siblings. A tree contains one node without father, termed *root*.

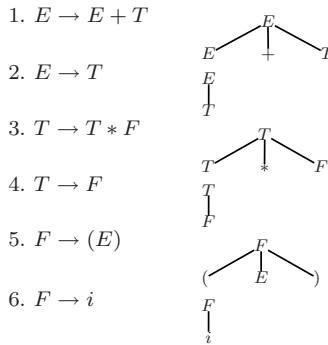
Consider an internal node  $N$ : the *subtree* with root  $N$  is the tree having  $N$  as root and containing all siblings of  $N$ , all of their siblings, etc., that is, all *descendants* of  $N$ . Nodes without sibling are termed *leaves* or *terminal nodes*. The sequence of all leaves, read from left to right, is the *frontier* of the tree.

A *syntax tree* has as root the axiom and as frontier a sentence.

To construct the tree consider a derivation. For each rule  $A_0 \rightarrow A_1 A_2 \dots A_r$ ,  $r \geq 1$  used in the derivation, draw a small tree having  $A_0$  as root and siblings  $A_1 A_2 \dots A_r$ , which may be terminals or nonterminals. If the rule is  $A_0 \rightarrow \varepsilon$ , draw one sibling labelled with epsilon. Such trees are then pasted together, by uniting each nonterminal sibling, say  $A_i$ , with the root node having the same label  $A_i$ , which is used to expand  $A_i$  in the subsequent step of the derivation.

*Example 2.40.* Syntax tree

The grammar is reproduced in Figure 2.1 numbering the rules for reference in the construction of the syntax tree.

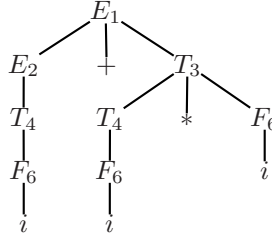


**Fig. 2.1** Grammar rules and corresponding tree fragments.

The derivation

$$E \xRightarrow{1} E+T \xRightarrow{2} T+T \xRightarrow{4} F+T \xRightarrow{6} i+T \xRightarrow{3} i+T*F \xRightarrow{4} i+F*F \xRightarrow{6} i+i*F \xRightarrow{6} i+i*i \quad (2.2)$$

corresponds to the following syntax tree:



where the labels of the rules applied are displayed. Notice the same tree represents other equivalent derivations, like the next one

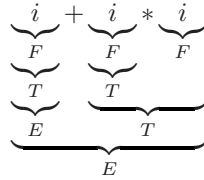
$$E \xRightarrow{1} E+T \xRightarrow{3} E+T*F \xRightarrow{6} E+T*i \xRightarrow{4} E+F*i \xRightarrow{6} E+i*i \xRightarrow{2} T+i*i \xRightarrow{4} F+i*i \xRightarrow{6} i+i*i \quad (2.3)$$

and many others which differ in the order rules are applied. Derivation (2.2) is termed *left* and derivation (2.3) is termed *right*.

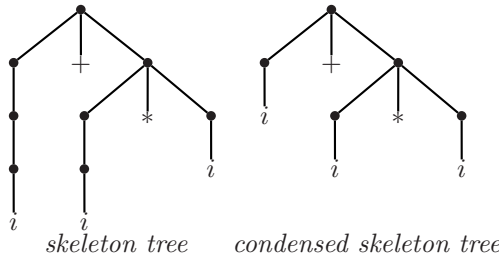
A syntax tree of a sentence  $x$  can also be encoded in a text, by enclosing each subtree between brackets<sup>18</sup>. Brackets are subscripted with the nonterminal symbol. Thus the preceding tree is encoded by the *parenthesized expression*

$$\left[ \left[ \left[ [i]_F \right]_T \right]_E + \left[ \left[ [i]_F \right]_T * [i]_F \right]_T \right]_E$$

or by



The representation can be simplified by dropping the nonterminal labels, thus obtaining a *skeleton tree* (left):



or the corresponding parenthesized string:

$$\left[ \left[ \left[ [i] \right] \right] + \left[ \left[ [i] \right] * [i] \right] \right]$$

<sup>18</sup> Assuming brackets not to be in the terminal alphabet.

A further simplification of the skeleton tree consists in shortening non-bifurcating paths, resulting in the *condensed skeleton tree* (right). The nodes fused together represent copy rules of the grammar. The corresponding parenthesized sentence is

$$[[i] + [[i] * [i]]]$$

Some approaches tend to view a grammar as a device for assigning structure to sentences. From this standpoint a grammar defines a set of syntax trees, that is, a tree language instead of a string language.<sup>19</sup>

## Left and Right Derivations

A derivation

$$\beta_0 \Rightarrow \beta_1 \Rightarrow \dots \Rightarrow \beta_p$$

where

$$\beta_i = \delta_i A_i \eta_i \text{ and } \beta_{i+1} = \delta_i \alpha_i \eta_i$$

is called *right* or rightmost (resp. *left*) if, for all  $0 \leq i \leq p-1$ , it is  $\eta_i \in \Sigma^*$  (resp.  $\delta_i \in \Sigma^*$ ).

In words, a right (resp. left) derivation expands at each step the rightmost (resp. leftmost) nonterminal. A letter  $r$  or  $l$  may be subscripted to the arrow sign, to make explicit the order of the derivation.

Observe that other derivations exist which are neither right nor left, because the nonterminal symbol expanded is not always either rightmost or leftmost, or because it is at some step rightmost and at some other step leftmost.

Returning to the preceding example, the derivation (2.2) is leftmost and is denoted by  $E \xRightarrow[l]{+} i + i * i$ . The derivation 2.3 is rightmost, whereas the derivation

$$\begin{aligned} E &\xRightarrow[l,r]{} E + T \xRightarrow[r]{} E + T * F \xRightarrow[l]{} T + T * F \xRightarrow{} T + F * F \xRightarrow[r]{} \\ &T + F * i \xRightarrow[l]{} F + F * i \xRightarrow[r]{} F + i * i \xRightarrow[r]{} i + i * i \end{aligned} \quad (2.4)$$

is neither right nor left. The three derivations are represented by the same tree.

This example actually illustrates an essential property of context-free grammars.

*Property 2.41.* Every sentence of a context-free grammar can be generated by a left (or right) derivation.

Therefore it does no harm to use just right (or left) derivations in the definition (p. 36) of the language generated by a grammar.

---

<sup>19</sup> A reference to the theory of tree languages is [21].

On the other hand, other more complex types of grammars, as the context-sensitive ones, do not share this nice property, which is quite important for obtaining efficient algorithms for string recognition and parsing, as we shall see.

### 2.5.8 Parenthesis Languages


Many artificial languages include parenthesized or nested structures, made by matching pairs of *opening/closing marks*. Any such occurrence may contain other matching pairs.

The marks are abstract elements that have different concrete representations in distinct settings. Thus Pascal block structures are enclosed within ‘begin’ ... ‘end’, while in language C curly brackets are used.

A massive use of parenthesis structures characterizes the mark-up language XML, that offers the possibility of inventing new matching pairs. An example is  $\langle title \rangle \dots \langle /title \rangle$  used to delimit the document title. Similarly, in the LaTeX notation this book was written in, a mathematical formula is enclosed between the marks  $\backslash begin\{equation\} \dots \backslash end\{equation\}$ .

When a marked construct may contain another construct of the same kind, it is called *self-nested*. Self-nesting is potentially unbounded in artificial languages, whereas in natural languages its use is moderate, because it causes difficulty of comprehension by breaking the flow of discourse. Next comes an example of a complex German sentence<sup>20</sup> with three nested relative clauses:

*der Mann der die Frau die das Kind das die Katze füttert sieht liebt schläft*



Abstracting from concrete representation and content, this paradigm is known as *Dyck language*. The terminal alphabet contains one or more pairs of opening/closing marks. An example is alphabet  $\Sigma = \{ ' , ' ( , ' ) , ' [ , ' ] \}$  and sentence  $[] (([] ()))$ .

Dyck sentences are characterized by the following *cancellation rule* that checks parentheses are well nested: given a string, repeatedly substitute the empty string for a pair of adjacent matching parentheses

$$[] \Rightarrow \varepsilon \quad ( ) \Rightarrow \varepsilon$$

thus obtaining another string. Repeat until the transformation no longer applies; the original string is correct if, and only if, the last string is empty.

---

<sup>20</sup> The man who loves the woman (who sees the child (who feeds the cat)) sleeps.



*Example 2.42.* Dyck language

To aid the eye, we encode left parentheses as  $a, b, \dots$  and right parentheses as  $a', b', \dots$

With alphabet  $\Sigma = \{a, a', b, b'\}$ , the Dyck language is generated by grammar:

$$S \rightarrow aSa'S \mid bSb'S \mid \varepsilon$$

Notice we need nonlinear rules (the first two) to generate this language.

To see that, compare with language  $L_1$  of example 2.29 on p. 30. The latter, recoding its alphabet  $\{b, e\}$  as  $\{a, a'\}$ , is strictly included in the Dyck language, since  $L_1$  disallows any string with two or more nests, e.g.,

$$a \underbrace{a \underbrace{aa'}_{\text{nest 1}} a'}_{\text{nest 2}} a \underbrace{a \underbrace{aa'}_{\text{nest 3}} a'}_{\text{nest 4}} a'$$

Such sentences have a branching syntax tree that requires nonlinear rules for its derivation.

Another way of constraining the grammar to produce nested constructs, is to force each rule to be parenthesized.

**Definition 2.43.** Parenthesized grammar

Let  $G = (V, \Sigma, P, S)$  be a grammar with an alphabet  $\Sigma$  not containing parentheses. The parenthesized grammar  $G_p$  has alphabet  $\Sigma \cup \{ ' ( ' , ' ) ' \}$  and rules

$$A \rightarrow (\alpha) \text{ where } A \rightarrow \alpha \text{ is a rule of } G$$

The grammar is *distinctly parenthesized* if every rule has form

$$A \rightarrow ({}_A \alpha )_A \quad B \rightarrow ({}_B \alpha )_B$$

where  $({}_A$  and  $)_A$  are parentheses subscripted with the nonterminal name.

Clearly each sentence produced by such grammars exhibits parenthesized structure.

*Example 2.44.* Parenthesis grammar

The parenthesized version of the grammar for lists of palindromes (p. 31) is

$$\begin{array}{ll} list \rightarrow (pal, list) & pal \rightarrow () \\ list \rightarrow (pal) & pal \rightarrow (a \, pal \, a) \\ & pal \rightarrow (b \, pal \, b) \end{array}$$

The original sentence  $aa$  becomes the parenthesized sentence  $((a ( ) a))$ .

A notable effect of the presence of parentheses is to allow a simpler checking of string correctness, to be discussed in Chapter 4.

### 2.5.9 Regular Composition of Context-Free Languages

If the basic operations of regular languages, union, concatenation, and star, are applied to context-free languages, the result remains a member of the  $CF$  family, to be shown next.

Let  $G_1 = (\Sigma_1, V_1, P_1, S_1)$  and  $G_2 = (\Sigma_2, V_2, P_2, S_2)$  be the grammars defining languages  $L_1$  and  $L_2$ . We need the unrestrictive hypothesis that nonterminal sets are disjoint,  $V_1 \cap V_2 = \emptyset$ . Moreover, we stipulate that symbol  $S$ , to be used as axiom of the grammar under construction, is not used by either grammar,  $S \notin (V_1 \cup V_2)$ .

**Union:** The union  $L_1 \cup L_2$  is defined by the grammar containing the rules of both grammars, plus the initial rules  $S \rightarrow S_1 \mid S_2$ . In formulas, the grammar is

$$G = (\Sigma_1 \cup \Sigma_2, \{S\} \cup V_1 \cup V_2, \{S \rightarrow S_1 \mid S_2\} \cup P_1 \cup P_2, S)$$

**Concatenation:** The concatenation  $L_1 L_2$  is defined by the grammar containing the rules of both grammars, plus the initial rule  $S \rightarrow S_1 S_2$ . The grammar is

$$G = (\Sigma_1 \cup \Sigma_2, \{S\} \cup V_1 \cup V_2, \{S \rightarrow S_1 S_2\} \cup P_1 \cup P_2, S)$$

**Star:** The grammar  $G$  of the starred language  $(L_1)^*$  includes the rules of  $G_1$  and rules  $S \rightarrow SS_1 \mid \varepsilon$ .

**Cross:** From the identity  $L^+ = L.L^*$ , the grammar of the cross language could be written applying the concatenation construction to  $L$  and  $L^*$ , but it is better to produce the grammar directly. The grammar  $G$  of language  $(L_1)^+$  contains the rules of  $G_1$  and rules  $S \rightarrow SS_1 \mid S_1$ .

From all this we have:

*Property 2.45.* The family  $CF$  of context-free languages is closed by union, concatenation, star, and cross.

*Example 2.46.* Union of languages

The language

$$L = \{a^i b^i c^* \mid i \geq 0\} \cup \{a^* b^i c^i \mid i \geq 0\} = L_1 \cup L_2$$

contains sentences of the form  $a^i b^j c^k$  with  $i = j \vee j = k$ , such as

$$a^5 b^5 c^2, a^5 b^5 c^5, b^5 c^5$$

The rules for the component languages are straightforward:

$G_1$	$G_2$
$S_1 \rightarrow XC$	$S_2 \rightarrow AY$
$X \rightarrow aXb \mid \varepsilon$	$Y \rightarrow bYc \mid \varepsilon$
$C \rightarrow cC \mid \varepsilon$	$A \rightarrow aA \mid \varepsilon$

We just add alternatives  $S \rightarrow S_1 \mid S_2$ , to trigger derivation with either grammar.

A word of caution: if the nonterminal sets overlap, this construction produces a grammar that generates a language typically larger than the union. To see it, replace grammar  $G_2$  with the trivially equivalent grammar  $G''$ :

$$S'' \rightarrow AX \quad X \rightarrow bXc \mid \varepsilon \quad A \rightarrow aA \mid \varepsilon$$

Then the putative grammar of the union,  $\{S \rightarrow S_1 \mid S''\} \cup P_1 \cup P''$ , would also allow hybrid derivations using rules from both grammars, thus generating for instance *abcabc*, which is not in the union language.

Notably, property 2.45 holds for both families *REG* and *CF*, but only the former is closed by intersection and complement, to be seen later.

### Grammar of Mirror Language

Examining the effect of string reversal on the sentences of a *CF* language, one immediately sees the family is closed with respect to reversal (the same as family *REG*). Given a grammar, the rules generating the mirror language are obtained reversing every right part of a rule.

#### 2.5.10 Ambiguity

The common linguistic phenomenon of ambiguity in natural language shows up when a sentence has two or more meanings. Ambiguity is of two kinds, semantic or syntactic. Semantic ambiguity occurs in the clause **a hot spring**, where the noun denotes either a coil or a season. A case of syntactic (or structural) ambiguity is **half baked chicken**, having different meanings depending on the structure assigned:

[[half baked] chicken] or [half [baked chicken]].

Although ambiguity may cause misunderstanding in human communication, negative consequences are counteracted by availability of nonlinguistic clues for choosing the intended interpretation.

Artificial languages too can be ambiguous, but the phenomenon is less deep than in human languages. In most situations ambiguity is a defect to be removed or counteracted.

A sentence  $x$  defined by grammar  $G$  is syntactically *ambiguous*, if it is generated with two different syntax trees. Then the grammar too is called *ambiguous*.

*Example 2.47.* Consider again the language of arithmetic expressions of example 2.39, p. 40, but define it with a different grammar  $G'$  equivalent to the

previous one:

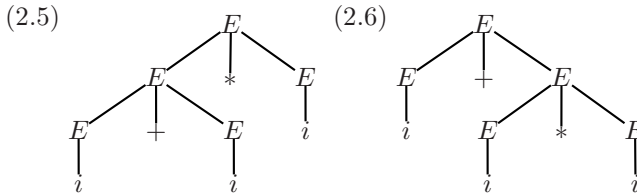
$$E \rightarrow E + E \mid E * E \mid (E) \mid i$$

The left derivations

$$E \Rightarrow E * E \Rightarrow E + E * E \Rightarrow i + E * E \Rightarrow i + i * E \Rightarrow i + i * i \quad (2.5)$$

$$E \Rightarrow E + E \Rightarrow i + E \Rightarrow i + E * E \Rightarrow i + i * E \Rightarrow i + i * i \quad (2.6)$$

generate the same sentence, with different trees:



The sentence  $i + i * i$  is therefore ambiguous.

Pay attention now to the meaning of the two readings of the same expression. The left tree interprets the sentence as  $(i + i) * i$ , the right tree assigns the interpretation  $i + (i * i)$ . The latter is likely to be preferable, because it agrees with the traditional precedence of operators. Another ambiguous sentence is  $i + i + i$ , having two trees that differ in the order of association of subexpressions: from left to right, or the other way. As a consequence grammar  $G'$  is ambiguous.

A major defect of this grammar is that it does not force the expected precedence of product over sum.

On the other hand, for grammar  $G$  of example 2.39 on p. 40 each sentence has only one left derivation, therefore all sentences are unambiguous, and the grammar as well.

It may be noticed that the new grammar  $G'$  is smaller than the old one  $G$ : this manifests a frequent property of ambiguous grammars, their conciseness with respect to equivalent unambiguous ones. In special situations, when one wants the simplest possible grammar for a language, ambiguity may be tolerated, but in general conciseness cannot be bought at the cost of equivocation.

The *degree of ambiguity* of a sentence  $x$  of language  $L(G)$  is the number of distinct syntax trees deriving the sentence. For a grammar the degree of ambiguity is the maximum degree of any ambiguous sentence. The next derivation shows such degree may be unbounded.

*Example 2.48.* (Example 2.47 continued)

The degree of ambiguity is 2 for sentence  $i + i + i$ ; it is 5 for  $i + i * i + i$ , as one sees from the skeleton trees:

$$\underbrace{i + i * i + i}_{\text{1}}, \quad \underbrace{i + i * i + i}_{\text{2}}, \quad \underbrace{i + i * i + i}_{\text{3}}, \quad \underbrace{i + i * i + i}_{\text{4}}, \quad \underbrace{i + i * i + i}_{\text{5}}$$

It is easy to see that longer sentences cause the degree of ambiguity to grow unbounded.

An important practical problem is to check a given grammar for ambiguity. This is an example of a seemingly simple problem, for which no general algorithm exists: the problem is undecidable.<sup>21</sup> This means that any general procedure for checking a grammar for ambiguity may be forced to examine longer and longer sentences, without ever reaching the certainty of the answer. On the other hand, for a specific grammar, with some ingenuity, one can often prove nonambiguity by applying some form of inductive reasoning.

In practice this is not necessary, and two approaches usually suffice. First, a small number of rather short sentences are tested, by constructing their syntax trees and checking that they are unique. If the test is passed, one has to check whether the grammar complies with certain conditions characterizing the so-called deterministic context-free languages, to be lengthily studied in Chapters 4 and 5. Such conditions are sufficient to ensure nonambiguity.

Even better is to prevent the problem when a grammar is designed, by avoiding some common pitfalls to be explained next.

### 2.5.11 Catalogue of Ambiguous Forms and Remedies

Following the definition, an ambiguous sentence displays two or more structures, each one possibly associated with a sensible interpretation. Though the cases of ambiguity are abundant in natural language, clarity of communication is not seriously impaired because sentences are uttered or written in a living context (gestures, intonation, presuppositions, etc.) that helps in selecting the interpretation intended by the author. On the contrary, in artificial languages ambiguity cannot be tolerated because machines are not as good as humans in making use of context, with the negative consequence of unpredictable behavior of the interpreter or compiler.

Now we classify the most common types of ambiguity and we show how to remove them by modifying the grammar, or in some cases the language.

#### Ambiguity from Bilateral Recursion

A nonterminal symbol  $A$  is bilaterally recursive if it is both left and right-recursive (i.e., it offers derivations  $A \Rightarrow^+ A\gamma$  and  $A \Rightarrow^+ \beta A$ ). We distinguish the case the two derivations are produced by the same or by different rules.

---

<sup>21</sup> A proof can be found in [28].

*Example 2.49.* Bilateral recursion from same rule

The grammar  $G_1$ :

$$E \rightarrow E + E \mid i$$

generates string  $i + i + i$  with two different left derivations:

$$\begin{aligned} E &\Rightarrow E + E \Rightarrow E + E + E \Rightarrow i + E + E \Rightarrow i + i + E \Rightarrow i + i + i \\ E &\Rightarrow E + E \Rightarrow i + E \Rightarrow i + E + E \Rightarrow i + i + E \Rightarrow i + i + i \end{aligned}$$

Ambiguity comes from the absence of a fixed order of generation of the string, from the left or from the right. Looking at the intended meaning as arithmetic formulas, this grammar does not specify the order of application of operations.

In order to remove ambiguity, observe this language is a list with separators  $L(G_1) = i(+i)^*$ , a paradigm we are able to define with a right-recursive rule,  $E \rightarrow i + E \mid i$ ; or with a left-recursive rule  $E \rightarrow E + i \mid i$ . Both are unambiguous.

*Example 2.50.* Left and right recursions in different rules

A second case of bilateral recursive ambiguity is grammar  $G_2$ :

$$A \rightarrow aA \mid Ab \mid c$$

This language too is regular:  $L(G_2) = a^*cb^*$ . It is the concatenation of two lists,  $a^*$  and  $b^*$ , with  $c$  interposed. Ambiguity disappears if the two lists are derived by separate rules, thus suggesting the grammar:

$$S \rightarrow AcB \quad A \rightarrow aA \mid \varepsilon \quad B \rightarrow bB \mid \varepsilon$$

An alternative remedy is to decide the first list should be generated before the second one (or conversely):

$$S \rightarrow aS \mid X \quad X \rightarrow Xb \mid c$$

Remark: a double recursion on the same nonterminal by itself does not cause ambiguity, if the two recursions are not left and right. Observe the grammar

$$S \rightarrow +SS \mid \times SS \mid i$$

that defines so-called prefix polish expressions with signs of sum and product (further studied in Chapter 6), such as  $++ii \times ii$ . Although two rules are doubly recursive, since one recursion is right but the other is not left, the grammar is not ambiguous.

## Ambiguity from Union

If languages  $L_1 = L(G_1)$  and  $L_2 = L(G_2)$  share some sentences, that is, their intersection is not empty, the grammar  $G$  of the united languages, constructed as explained on p. 46, is ambiguous. (No need to repeat the two component grammars should have disjoint nonterminal sets.)

Take a sentence  $x \in L_1 \cap L_2$ . It is obviously produced by two distinct derivations, one using rules of  $G_1$ , the other using rules of  $G_2$ . The sentence is ambiguous for grammar  $G$  that contains all the rules. Notice a sentence  $x$  belonging to the first but not the second language,  $x \in L_1 \setminus L_2$ , is derived by rules of  $G_1$  only, hence is not ambiguous (if the first grammar is so).

*Example 2.51.* Union of overlapping languages

In language and compiler design there are various causes for overlap.

1. When one wants to single out a special pattern requiring special processing, within a general class of phrases. Consider additive arithmetic expressions with constants  $C$  and variables  $i$ . A grammar is

$$E \rightarrow E + C \mid E + i \mid C \mid i \quad C \rightarrow 0 \mid 1D \mid \dots \mid 9D \quad D \rightarrow 0D \mid \dots \mid 9D \mid \varepsilon$$

Now assume the compiler has to single out such expressions as  $i + 1$  or  $1 + i$ , because they have to be translated to machine code, using increment instead of addition. To this end we add the rules

$$E \rightarrow i + 1 \mid 1 + i$$

Unfortunately, the new grammar is ambiguous, since a sentence like  $1 + i$  is generated by the original rules too.

2. When the same operator is overloaded, i.e. used with different meanings in different constructs. In language Pascal the sign ‘+’ denotes both addition in

$$E \rightarrow E + T \mid T \quad T \rightarrow V \quad V \rightarrow \dots$$

and set union in

$$E_{set} \rightarrow E_{set} + T_{set} \mid T_{set} \quad T_{set} \rightarrow V$$

Such ambiguities need severe grammar surgery to be eliminated: either the two ambiguous constructs are made disjoint or they are fused together. Disjunction of constructs is not feasible in the previous examples, because string ‘1’ cannot be removed from the set of integer constants derived from nonterminal  $C$ . To enforce a special treatment of value ‘1’, if one accepts a syntactic change to the language, it suffices to add operator *inc* (for increment by 1) and replace rule  $E \rightarrow i + 1 \mid 1 + i$  with rule  $E \rightarrow inc\ i$ .

In the latter example ambiguity is semantic, caused by the double meaning (polysemy) of operator ‘+’. A remedy is to collapse together the rules for arithmetic expressions (generated from  $E$ ) and set expressions (generated

from  $E_{set}$ ), thus giving up a syntax-based separation. The semantic analyzer will take care of it. Alternatively, if modifications are permissible, one may replace ‘+’ by character ‘ $\cup$ ’ in set expressions.

In the following examples removal of overlapping constructs actually succeeds.

*Example 2.52.* (McNaughton)

1. Grammar  $G$ :

$$S \rightarrow bS \mid cS \mid D \quad D \rightarrow bD \mid cD \mid \varepsilon$$

is ambiguous since  $L(G) = \{b, c\}^* = L_D(G)$ . The derivations

$$S \xRightarrow{+} bbcD \Rightarrow bbc \quad S \Rightarrow D \xRightarrow{+} bbcD \Rightarrow bbc$$

produce the same result. Deleting the rules of  $D$ , which are redundant, we have  $S \rightarrow bS \mid cS \mid \varepsilon$ .

2. Grammar

$$S \rightarrow B \mid D \quad B \rightarrow bBc \mid \varepsilon \quad D \rightarrow dDe \mid \varepsilon$$

where  $B$  generates  $b^n c^n, n \geq 0$ , and  $D$  generates  $d^n e^n, n \geq 0$ , has just one ambiguous sentence:  $\varepsilon$ . Remedy: generate it directly from axiom:

$$S \rightarrow B \mid D \mid \varepsilon \quad B \rightarrow bBc \mid bc \quad D \rightarrow dDe \mid de$$

## Ambiguity from Concatenation

Concatenating languages may cause ambiguity, if a suffix of a sentence of language one is also a prefix of a sentence of language two.

Remember the grammar  $G$  of concatenation  $L_1 L_2$  (p. 46) contains rule  $S \rightarrow S_1 S_2$  in addition to the rules of  $G_1$  and  $G_2$  (by hypothesis not ambiguous). Ambiguity arises in  $G$  if the following sentences exist in the languages:

$$u' \in L_1 \quad u'v \in L_1 \quad vz'' \in L_2 \quad z'' \in L_2$$

Then string  $u'vz''$  of language  $L_1.L_2$  is ambiguous, via the derivations

$$S \Rightarrow S_1 S_2 \xRightarrow{+} u' S_2 \xRightarrow{+} u' v z'' \quad S \Rightarrow S_1 S_2 \xRightarrow{+} u' v S_2 \xRightarrow{+} u' v z''$$

*Example 2.53.* Concatenation of Dyck languages

For the concatenation  $L = L_1 L_2$  of the Dyck languages (p. 44)  $L_1$  and  $L_2$  over alphabets (in the order given)  $\{a, a', b, b'\}$  and  $\{b, b', c, c'\}$ , a sentence is  $aa'bb'cc'$ . The standard grammar of  $L$  is

$$S \rightarrow S_1 S_2 \quad S_1 \rightarrow aS_1 a' S_1 \mid bS_1 b' S_1 \mid \varepsilon \quad S_2 \rightarrow bS_2 b' S_2 \mid cS_2 c' S_2 \mid \varepsilon$$

The sentence is derived in two ways:



$$\begin{array}{cc} \overbrace{aa'bb'}^{S_1} & \overbrace{cc'}^{S_2} & \overbrace{aa'}^{S_1} & \overbrace{bb'cc'}^{S_2} \end{array}$$

To remove this ambiguity one should block the movement of a string from the suffix of language one to the prefix of language two (and conversely).

If the designer is free to modify the language, a simple remedy is to interpose a new terminal as separator between the two languages. In our example, with  $\#$  as separator, the language  $L_1\#L_2$  is easily defined without ambiguity by a grammar with initial rule  $S \rightarrow S_1\#S_2$ .

Otherwise, a more complex unambiguous solution would be to write a grammar, with the property that any string not containing  $c$ , such as  $bb'$ , is in language  $L_1$  but not in  $L_2$ . Notice that string  $bcc'b'$  is on the other hand, assigned to language two.

## Unique Decoding

A nice illustration of concatenation ambiguity comes from the study of codes in information theory. An information source is a process producing a message, i.e., a sequence of symbols from a finite set  $\Gamma = \{A, B, \dots, Z\}$ . Each such symbol is then encoded into a string over a terminal alphabet  $\Sigma$  (typically binary); a coding function maps each symbol into a short terminal string, termed its code.

Consider for instance the following source symbols and their mapping into binary codes ( $\Sigma = \{0, 1\}$ ):

$$\Gamma = \left\{ \overbrace{A}^{01}, \overbrace{C}^{10}, \overbrace{E}^{11}, \overbrace{R}^{001} \right\}$$

Message *ARRECA* is encoded as 01 001 001 11 10 01; by decoding this string the original text is the only one to be obtained. Message coding is expressed by grammar  $G_1$ :

$$Mess \rightarrow A Mess \mid C Mess \mid E Mess \mid R Mess \mid A \mid C \mid E \mid R$$

$$A \rightarrow 01 \quad C \rightarrow 10 \quad E \rightarrow 11 \quad R \rightarrow 001$$

The grammar generates a message such as *AC* by concatenating the corresponding codes, as displayed in the syntax tree:

$$\begin{array}{cc} & \overbrace{\hspace{1.5cm}}^{Mess} \\ & \overbrace{\hspace{1.5cm}}^{Mess} \\ \overbrace{A} & \overbrace{C} \\ \overbrace{01} & \overbrace{10} \end{array}$$

As the grammar is clearly unambiguous, every encoded message, i.e., every sentence of language  $L(G_1)$ , has one and only one syntax tree corresponding to the decoded message.

On the contrary, the next bad choice of codes

$$\Gamma = \{ \overbrace{A}^{00}, \overbrace{C}^{01}, \overbrace{E}^{10}, \overbrace{R}^{010} \}$$

renders ambiguous the grammar

$$Mess \rightarrow A Mess \mid C Mess \mid E Mess \mid R Mess \mid A \mid C \mid E \mid R$$

$$A \rightarrow 00 \quad C \rightarrow 01 \quad E \rightarrow 10 \quad R \rightarrow 010$$

Consequently, message 00010010100100 can be deciphered in two ways, as *ARRECA* or as *ACAECCA*.

The defect has two causes: the identity

$$\underbrace{01}_{\text{first}}.00.10 = \underbrace{010}_{\text{first}}.010$$

holds for two pairs of concatenated codes; and the first codes, 01 and 010, are one prefix of the other.

Code theory studies these and similar conditions that make a set of codes uniquely decipherable.

### Other Ambiguous Situations

The next case is similar to the ambiguity of regular expressions (p. 22).

*Example 2.54.* Consider the grammar

$$S \rightarrow DcD \quad D \rightarrow bD \mid cD \mid \varepsilon$$

Rule one says a sentence contains at least one  $c$ ; the alternatives of  $D$  generate  $\{b, c\}^*$ . The same language structure would be defined by regular expression  $\{b, c\}^* c \{b, c\}^*$ , which is ambiguous: every sentence with two or more  $c$  is ambiguous since the distinguished occurrence of  $c$  is not fixed. This defect can be repaired imposing the distinguished  $c$  is, say, the leftmost one:

$$S \rightarrow BcD \quad D \rightarrow bD \mid cD \mid \varepsilon \quad B \rightarrow bB \mid \varepsilon$$

Notice that  $B$  may not derive a string containing  $c$ .

*Example 2.55.* Setting an order on rules

In grammar

$$S \rightarrow bSc \mid bbSc \mid \varepsilon$$

the first two rules may be applied in one or the other order, producing the ambiguous derivations

$$S \Rightarrow bbSc \Rightarrow bbbScc \Rightarrow bbbcc \quad S \Rightarrow bSc \Rightarrow bbbScc \Rightarrow bbbcc$$

Remedy: oblige rule one to precede rule two:

$$S \rightarrow bSc \mid D \quad D \rightarrow bbDc \mid \varepsilon$$

### Ambiguity of Conditional Statements

A notorious case of ambiguity in programming languages with conditional instructions occurred in the first version of language Algol 60,<sup>22</sup> a milestone for applications of *CF* grammars. Consider grammar

$$S \rightarrow \text{if } b \text{ then } S \text{ else } S \mid \text{if } b \text{ then } S \mid a$$

where *b* stands for a boolean condition and *a* for any nonconditional instruction, both left undefined for brevity. The first alternative produces a *two legs* conditional instruction, the second a *one-leg* construct.

Ambiguity arises when two sentences are nested, with the outermost being a two-way conditional. For instance, examine the two readings:

$$\overbrace{\text{if } b \text{ then } \underbrace{\text{if } b \text{ then } a \text{ else } a}} \quad \overbrace{\text{if } b \text{ then } \underbrace{\text{if } b \text{ then } a}} \text{ else } a$$

caused by the “dangling” *else*.

It is possible to eliminate the ambiguity at the cost of complicating the grammar. Assume we decide to choose the left skeleton tree, that binds the *else* to the immediately preceding *if*. The corresponding grammar is:

$$S \rightarrow S_E \mid S_T \quad S_E \rightarrow \text{if } b \text{ then } S_E \text{ else } S_E \mid a$$

$$S_T \rightarrow \text{if } b \text{ then } S_E \text{ else } S_T \mid \text{if } b \text{ then } S$$

Observe the syntax class *S* has been split into two classes: *S<sub>E</sub>* defines a two-legs conditional such that its nested conditionals are in the same class *S<sub>E</sub>*. The other syntax class *S<sub>T</sub>* defines a one-leg conditional, or a two-legs conditional such that the first nested conditional is of class *S<sub>E</sub>* and the second is of class *S<sub>T</sub>*; this excludes the combinations

$$\text{if } b \text{ then } S_T \text{ else } S_T \quad \text{and} \quad \text{if } b \text{ then } S_T \text{ else } S_E$$

The facts that only *S<sub>E</sub>* may precede *else*, and that only *S<sub>T</sub>* defines a one-leg conditional, disallow derivation of the right skeleton tree.

<sup>22</sup> The following official version [39] removed the ambiguity.

If the language syntax can be modified, a simpler solution exists: many language designers have introduced a closing mark for delimiting conditional constructs. See the next use of mark *end\_if*:

$$S \rightarrow \text{if } b \text{ then } S \text{ else } S \text{ end\_if} \mid \text{if } b \text{ then } S \text{ end\_if} \mid a$$

Indeed this is a sort of parenthesizing (p. 45) of the original grammar.

## Inherent Ambiguity of Language

In all preceding examples we have found that a language is defined by equivalent grammars, some ambiguous some not. But this is not always the case. A language is called *inherently ambiguous* if every grammar of the language is ambiguous. Surprisingly enough, inherently ambiguous languages exist!

*Example 2.56.* Unavoidable ambiguity from union

Recall example 2.46 on p. 46:

$$L = \{a^i b^j c^k \mid (i, j, k \geq 0) \wedge ((i = j) \vee (j = k))\}$$

The language can be equivalently defined by means of the union

$$L = \{a^i b^i c^* \mid i \geq 0\} \cup \{a^* b^i c^i \mid i \geq 0\} = L_1 \cup L_2$$

of two nondisjoint languages.

We intuitively argue that any grammar of this language is necessarily ambiguous. The grammar on p. 46 unites the rules of the component grammars, and is obviously ambiguous for every sentence  $x \in \{\varepsilon, abc, \dots a^i b^i c^i \dots\}$ , shared by both languages. Any such sentence is produced by  $G_1$  using rules checking that  $|x|_a = |x|_b$ , a check only possible for a syntax structure of the type

$$\underbrace{a \dots a \quad \underbrace{ab} \quad b \dots b \quad cc \dots c}_{\text{structure of type } L_1}$$

Now the same string  $x$ , viewed as a sentence of  $L_2$ , must be generated with a structure of type

$$\underbrace{a \dots aa \quad b \dots b \quad \underbrace{bc} \quad c \dots c}_{\text{structure of type } L_2}$$

in order to perform the equality check  $|x|_b = |x|_c$ .

No matter which variation of the grammar we make, the two equality checks on the exponents are unavoidable for such sentences and the grammar remains ambiguous.

In reality, inherent language ambiguity is rare and hardly, if ever, affects technical languages.

2.5.12 Weak and Structural Equivalence

It is not enough for a grammar to generate the correct sentences; it should also assign to each one a suitable structure, in agreement with the intended meaning. This requirement of *structural adequacy* has already been invoked at times, for instance when discussing operator precedence in hierarchical lists.

We ought to reexamine the notion of grammar in the light of structural adequacy. Recall the definition on p. 36: two grammars are equivalent if they define the same language,  $L(G) = L(G')$ . Such definition, to be qualified now as *weak equivalence*, poorly fits with the real possibility of substituting one grammar for the other in technical artifacts such as compilers. The reason is the two grammars are not guaranteed to assign the same meaningful structure to every sentence.

We need a more stringent definition of equivalence, which is only relevant for unambiguous grammars. Grammars  $G$  and  $G'$  are *strongly or structurally equivalent*, if  $L(G) = L(G')$  and in addition  $G$  and  $G'$  assign to each sentence two syntax trees, which may be considered *structurally similar*.

The last condition should be formulated in accordance with the intended application. A plausible formulation is: two syntax trees are structurally similar if the corresponding condensed skeleton trees (p. 43) are equal.

Strong equivalence implies weak equivalence, but the former is a decidable property, unlike the latter.<sup>23</sup>

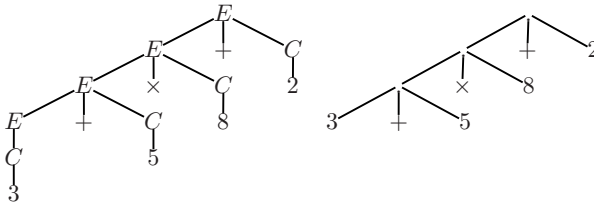
*Example 2.57.* Structural adequacy of arithmetic expressions

The difference between strong and weak equivalence is manifested by the case of arithmetic expressions, such as  $3 + 5 \times 8 + 2$ , first viewed as a list of digits separated by plus and times signs.

- First grammar  $G_1$ :

$$\begin{array}{l} E \rightarrow E + C \quad E \rightarrow E \times C \quad E \rightarrow C \\ C \rightarrow 0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9 \end{array}$$

The syntax tree of the previous sentence is



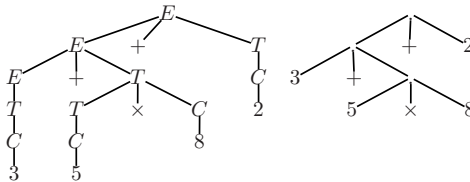
In the condensed skeleton (right), nonterminals and copy rules have been dropped.

<sup>23</sup> The decision algorithm, e.g., in [47], is similar to the one for checking the equivalence of finite automata, to be presented in next chapter.

- A second grammar  $G_2$  for this language is

$$\begin{aligned} E &\rightarrow E + T & E &\rightarrow T & T &\rightarrow T \times C & T &\rightarrow C \\ C &\rightarrow 0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9 \end{aligned}$$

The two grammars are weakly equivalent. Observing now the syntax tree of the same sentence:



we see its skeleton differs from the previous one: it contains a subtree with frontier  $5 \times 8$ , associated with a multiplication. Therefore the grammars are not structurally equivalent.

Is either one of the grammars preferable? Concerning ambiguity, both grammars are all right. But only grammar  $G_2$  is structurally adequate, if one considers also meaning. In fact, sentence  $3+5 \times 8+2$  denotes a computation, to be executed in the traditional order:  $3+(5 \times 8)+2 = (3+40)+2 = (43+2) = 45$ : this is the *semantic interpretation*. The parentheses specifying the order of evaluation  $((3 + (5 \times 8)) + 2)$  can be mapped on the subtrees of the skeletal tree produced by  $G_2$ .

On the contrary, grammar  $G_1$  produces the parenthesizing  $((3 + 5) \times 8) + 2$  which is not adequate, because by giving precedence to the first sum over the product, it returns the wrong semantic interpretation, 66.

Incidentally, the second grammar is more complex because enforcing operator precedence requires more nonterminals and rules.

It is crucial for a grammar intended for driving a compiler to be structurally adequate, as we shall see in the last chapter on syntax-directed translations.

- A case of structural equivalence is illustrated by the next grammar  $G_3$ <sup>24</sup>:

$$\begin{aligned} E &\rightarrow E + T \mid T + T \mid C + T \mid E + C \mid T + C \mid C + C \mid T \times C \mid C \times C \mid C \\ T &\rightarrow T \times C \mid C \times C \mid C \\ C &\rightarrow 0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9 \end{aligned}$$

Now the condensed skeleton trees of any arithmetic expression coincide for grammars  $G_2$  and  $G_3$ . They are structurally equivalent.

<sup>24</sup> This grammar has more rules because it does not exploit copy rules to express inclusion of syntax classes. Categorization and ensuing taxonomies reduce the complexity of descriptions in any area of knowledge.

## Generalized Structural Equivalence

Sometimes a looser criterion of similarity than strict identity of condensed skeleton trees is more suitable. This consists in requesting that the two corresponding trees should be easily mapped one on the other by some simple transformation. The idea can be differently materialized: one possibility is to have a bijective correspondence between the subtrees of one tree and the subtrees of the other. For instance, the grammars

$$\{S \rightarrow Sa \mid a\} \quad \text{and} \quad \{X \rightarrow aX \mid a\}$$

are just weakly equivalent in generating  $L = a^+$ , since the condensed skeleton trees differ, as in the example of sentence  $aa$ :

$$\underbrace{a} \quad a \quad \text{and} \quad a \quad \underbrace{a}$$

However, the two grammars may be considered structurally equivalent in a generalized sense because each left-linear tree of the first grammar corresponds to a right-linear tree of the second. The intuition that the two grammars are similar is satisfied because their trees are specularly identical, i.e., they become coincident by turning left-recursive rules into right-recursive (or conversely).

### 2.5.13 Grammar Transformations and Normal Forms

We are going to study a range of transformations that are useful to obtain grammars having certain desired properties, without affecting the language. Normal forms are restricted rule patterns, yet allowing any context-free language to be defined. Such forms are widely used in theoretical papers, to simplify statements and proofs of theorems. Otherwise, in applied work, grammars in normal form are rarely used because they are much larger and less readable.

Several grammar transformations (e.g., the movement of recursion from left to right) are useful for parsing algorithms. We start the survey of transformations from simple ones.

Let grammar  $G = (V, \Sigma, P, S)$  be given.

#### Nonterminal Expansion

A general-purpose transformation preserving language is *nonterminal expansion*, consisting of replacing a nonterminal with its alternatives.

Replace rule  $A \rightarrow \alpha B \gamma$  with rules

$$A \rightarrow \alpha\beta_1\gamma \mid \alpha\beta_2\gamma \mid \dots \mid \alpha\beta_n\gamma$$

where  $B \rightarrow \beta_1 \mid \beta_2 \mid \dots \mid \beta_n$  are all the alternatives of  $B$ . Clearly the language does not change, since the two-step derivation  $A \Rightarrow \alpha B \gamma \Rightarrow \alpha\beta_i\gamma$  becomes the immediate derivation  $A \Rightarrow \alpha\beta_i\gamma$ , to the same effect.

### Axiom Elimination from Right Parts

At no loss of generality, every right part of a rule may exclude the presence of the axiom, i.e., be a string over alphabet  $(\Sigma \cup (V \setminus \{S\}))$ . To this end, just introduce a new axiom  $S_0$  and rule  $S_0 \rightarrow S$ .

### Nullable Nonterminals and Elimination of Empty Rules

A nonterminal  $A$  is *nullable* if it can derive the empty string, i.e.,  $A \xRightarrow{+} \varepsilon$ . Consider the set named  $Null \subseteq V$  of nullable nonterminals. The set is computed by the following logical clauses, to be applied in any order until the set ceases to grow, i.e., a fixed point is reached:

$$\begin{aligned} A \in Null & \text{ if } A \rightarrow \varepsilon \in P \\ A \in Null & \text{ if } (A \rightarrow A_1 A_2 \dots A_n \in P \text{ with } A_i \in V \setminus \{A\}) \wedge \forall A_i (A_i \in Null) \end{aligned}$$

Row one finds the nonterminals that are immediately nullable; row two finds those which derive a string of nullable nonterminals.

*Example 2.58.* Computing nullable nonterminals

$$S \rightarrow SAB \mid AC \quad A \rightarrow aA \mid \varepsilon \quad B \rightarrow bB \mid \varepsilon \quad C \rightarrow cC \mid c$$

Result:  $Null = \{A, B\}$ .

Notice that, if rule  $S \rightarrow AB$  were added to the grammar, the result would include  $S \in Null$ .

The *normal form without nullable nonterminals*, for brevity *nonnullable*, is defined by the condition that no nonterminal other than the axiom is nullable. Moreover, the axiom is nullable if, and only if, the empty string is in the language.

To construct the non-nullable form, first compute the set  $Null$ , then do as follows:

- for each rule  $A \rightarrow A_1 A_2 \dots A_n \in P$ , with  $A_i \in V \cup \Sigma$ , add as alternatives the strings obtained from the right part, deleting in all possible ways any nullable nonterminal  $A_i$ ;
- remove the empty rules  $A \rightarrow \varepsilon$ , for every  $A \neq S$ .

If the grammar thus obtained is unclear or circular, it should be cleaned with the known algorithms (p. 37).



*Example 2.59.* (Example 2.58 continued)

In Table 2.4, column one lists nullability. The other columns list the original rules, those produced by the transformation, and the final clean rules.

**Table 2.4** Elimination of copy rules.

<i>Nullable</i>	<i>G original</i>	<i>G' to be cleaned</i>	<i>G' nonnullable normal</i>
<i>F</i>	$S \rightarrow SAB$	$S \rightarrow SAB \mid SA \mid SB \mid S$	$S \rightarrow SAB \mid SA \mid SB$
	$AC$	$\mid AC \mid C$	$\mid AC \mid C$
<i>V</i>	$A \rightarrow aA \mid \varepsilon$	$A \rightarrow aA \mid a$	$A \rightarrow aA \mid a$
<i>V</i>	$B \rightarrow bB \mid \varepsilon$	$B \rightarrow bB \mid b$	$B \rightarrow bB \mid b$
<i>F</i>	$C \rightarrow cC \mid c$	$C \rightarrow cC \mid c$	$C \rightarrow cC \mid c$

## Copies or Subcategorization Rules and Their Elimination

A *copy* (or *subcategorization*) rule has the form  $A \rightarrow B$ , with  $B \in V$ , a nonterminal symbol. Any such rule is tantamount as the relation  $L_B(G) \subseteq L_A(G)$ : the syntax class  $B$  is included in the class  $A$ .

For a concrete example, the rules

$$iterative\_phrase \rightarrow while\_phrase \mid for\_phrase \mid repeat\_phrase$$

introduce three subcategories of iterative phrases: **while**, **for**, and **repeat**. Although copy rules can be eliminated, many more alternatives have to be introduced and grammar legibility usually deteriorates. Notice that copy elimination reduces the height of syntax trees by shortening derivations.

For grammar  $G$  and nonterminal  $A$ , we define the set  $Copy(A) \subseteq V$  containing the nonterminals that are immediate or transitive copies of  $A$ :

$$Copy(A) = \{B \in V \mid \text{there is a derivation } A \xRightarrow{*} B\}$$

Note: if nonterminal  $C$  is nullable, the derivation may take the form

$$A \xRightarrow{+} BC \Rightarrow B$$

For simplicity we assume the grammar is nonnullable and the axiom does not occur in a right part.

To compute the set  $Copy$ , apply the following logical clauses until a fixed point is reached:

$$\begin{aligned} A &\in Copy(A) && \text{- initialization} \\ C &\in Copy(A) \text{ if } (B \in Copy(A)) \wedge (B \rightarrow C \in P) \end{aligned}$$

Then construct the rules  $P'$  of a new grammar  $G'$ , equivalent to  $G$  and copy-free, as follows:

$$P' := P \setminus \{A \rightarrow B \mid A, B \in V\} \quad \text{-- copy cancellation}$$

$$P' := \{A \rightarrow \alpha \mid \alpha \in ((\Sigma \cup V)^* \setminus V)\} \text{ where } (B \rightarrow \alpha) \in P \wedge B \in \text{Copy}(A)$$

The effect is that the old grammar derivation  $A \xRightarrow{\pm} B \Rightarrow \alpha$  shrinks to the immediate derivation  $A \Rightarrow \alpha$ .

Notice the transformation keeps all original noncopy rules. In Chapter 3 the same transformation will be applied to remove spontaneous moves from an automaton.

*Example 2.60.* Copy-free rules for arithmetic expressions

Applying the algorithm to grammar  $G_2$

$$\begin{array}{l} E \rightarrow E + T \mid T \quad T \rightarrow T \times C \mid C \\ C \rightarrow 0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9 \end{array}$$

we obtain

$$\text{Copy}(E) = \{E, T, C\}, \quad \text{Copy}(T) = \{T, C\}, \quad \text{Copy}(C) = \{C\}$$

The equivalent copy-free grammar is

$$\begin{array}{l} E \rightarrow E + T \mid T \times C \mid 0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9 \\ T \rightarrow T \times C \mid 0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9 \\ C \rightarrow 0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9 \end{array}$$

It is worth repeating that copy rules are very convenient for reusing certain blocks of rules, corresponding to syntactic subcategories; the grammar is concise and evidences the generalization and specialization of language constructs. For these reasons, reference manuals of technical languages cannot do without copy rules.

## Chomsky or Binary Normal Form

There are two types of rules:

1. *homogeneous binary*:  $A \rightarrow BC$ , where  $B, C \in V$
2. *terminal with singleton right part*:  $A \rightarrow a$ , where  $a \in \Sigma$

Moreover, if the empty string is in the language, there is rule  $S \rightarrow \varepsilon$  but the axiom is not allowed in any right part.

With such constraints any internal node of a syntax tree may have either two nonterminal siblings or one terminal son.

Given a grammar, by simplifying hypothesis without nullable nonterminals, we explain how to obtain a Chomsky normal form. Each rule  $A_0 \rightarrow A_1 A_2 \dots A_n$  of length  $n > 2$  is converted to a length 2 rule, singling out the first symbol  $A_1$  and the remaining suffix  $A_2 \dots A_n$ . Then a new ancillary nonterminal is created, named  $\langle A_2 \dots A_n \rangle$ , and the new rule

$$\langle A_2 \dots A_n \rangle \rightarrow A_2 \dots A_n$$

Now the original rule is replaced by

$$A_0 \rightarrow A_1 \langle A_2 \dots A_n \rangle$$

If symbol  $A_1$  is terminal, we write instead the rules:

$$A_0 \rightarrow \langle A_1 \rangle \langle A_2 \dots A_n \rangle \quad \langle A_1 \rangle \rightarrow A_1$$

where  $\langle A_1 \rangle$  is a new ancillary nonterminal.

Continue applying the same series of transformations to the grammar thus obtained, until all rules are in the form requested.

*Example 2.61.* Conversion to Chomsky normal form

The grammar

$$S \rightarrow dA \mid cB \quad A \rightarrow dAA \mid cS \mid c \quad B \rightarrow cBB \mid dS \mid d$$

becomes

$$\begin{aligned} S &\rightarrow \langle d \rangle A \mid \langle c \rangle B & A &\rightarrow \langle d \rangle \langle AA \rangle \mid \langle c \rangle S \mid c & B &\rightarrow \langle c \rangle \langle BB \rangle \mid \langle d \rangle S \mid d \\ \langle d \rangle &\rightarrow d & \langle c \rangle &\rightarrow c & \langle AA \rangle &\rightarrow AA & \langle BB \rangle &\rightarrow BB \end{aligned}$$

This form is used in mathematical essays, but rarely in practical work.

## Conversion of Left to Right Recursions

Another normal form termed *not left-recursive* is characterized by the absence of left-recursive rules or derivations (l-recursions); it is indispensable for the top-down parsers to be studied in Chapter 4. We explain how to transform l-recursions to right recursions.

### Transformation of Immediate l-Recursions

The more common and easier case is when the l-recursion to be eliminated is immediate. Consider the l-recursive alternatives of a nonterminal  $A$ :

$$A \rightarrow A\beta_1 \mid A\beta_2 \mid \dots \mid A\beta_h, h \geq 1$$

where no  $\beta_i$  is empty, and let

$$A \rightarrow \gamma_1 \mid \gamma_2 \mid \dots \mid \gamma_k, k \geq 1$$

be the remaining alternatives.

Create a new ancillary nonterminal  $A'$  and replace the previous rules with the next ones:

$$A \rightarrow \gamma_1 A' \mid \gamma_2 A' \mid \dots \mid \gamma_k A' \mid \gamma_1 \mid \gamma_2 \mid \dots \mid \gamma_k$$

$$A' \rightarrow \beta_1 A' \mid \beta_2 A' \mid \dots \mid \beta_h A' \mid \beta_1 \mid \beta_2 \mid \dots \mid \beta_h$$

Now every original l-recursive derivation, as for instance

$$A \Rightarrow A\beta_2 \Rightarrow A\beta_3\beta_2 \Rightarrow \gamma_1\beta_3\beta_2$$

is replaced with the equivalent right-recursive derivation

$$A \Rightarrow \gamma_1 A' \Rightarrow \gamma_1 \beta_3 A' \Rightarrow \gamma_1 \beta_3 \beta_2$$

*Example 2.62.* Converting immediate l-recursions to right recursion  
In the usual grammar of expressions

$$E \rightarrow E + T \mid T \quad T \rightarrow T * F \mid F \quad F \rightarrow (E) \mid i$$

nonterminals  $E$  and  $T$  are immediately l-recursive. Applying the transformation, the right-recursive grammar is obtained:

$$E \rightarrow TE' \mid T \quad E' \rightarrow +TE' \mid +T$$

$$T \rightarrow FT' \mid F \quad T' \rightarrow *FT' \mid *F \quad F \rightarrow (E) \mid i$$

Actually, in this case but not always, a simpler solution is possible, to specularly reverse the l-recursive rules, obtaining

$$E \rightarrow T + E \mid T \quad T \rightarrow F * T \mid F \quad F \rightarrow (E) \mid i$$

### Transformation of Nonimmediate Left Recursions

The next algorithm is used to transform nonimmediate l-recursions. We present it under the simplifying assumptions that grammar  $G$  is homogeneous, nonnullable, with singleton terminal rules; in other words, the rules are like in Chomsky normal form, but more than two nonterminals are permitted in a right part.

There are two nested loops; the external loop employs nonterminal expansion to change non-immediate into immediate l-recursions, thus shortening the length of derivations. The internal loop converts immediate l-recursions to right recursions; in so doing it creates ancillary nonterminals.

Let  $V = \{A_1, A_2, \dots, A_m\}$  be the nonterminal alphabet and  $A_1$  the axiom. For orderly scanning, we view the nonterminals as an (arbitrarily) ordered set, from 1 to  $m$ .

## Algorithm for Left Recursion Elimination

for  $i := 1$  to  $m$  do  
  for  $j := 1$  to  $i - 1$  do  
    replace every rule of type  $A_i \rightarrow A_j \alpha$ , where  $i > j$ , with the rules:  
 $A_i \rightarrow \gamma_1 \alpha \mid \gamma_2 \alpha \mid \dots \mid \gamma_k \alpha$   
    (- - possibly creating immediate l-recursions)  
    where  $A_j \rightarrow \gamma_1 \mid \gamma_2 \mid \dots \mid \gamma_k$  are the alternatives of nonterminal  $A_j$   
  end do  
  eliminate, by means of the previous algorithm, any immediate l-recursion  
  that may have arisen as alternative of  $A_i$ , creating the ancillary  
  nonterminal  $A'_i$   
end do

The idea<sup>25</sup> is to modify the rules in such a way that, if the right part of a rule  $A_i \rightarrow A_j \dots$  starts with a nonterminal  $A_j$ , then it is  $j > i$ , i.e., the latter nonterminal follows in the ordering.

*Example 2.63.* Applying the algorithm to grammar  $G_3$

$$A_1 \rightarrow A_2 a \mid b \quad A_2 \rightarrow A_2 c \mid A_1 d \mid e$$

which has the l-recursion  $A_1 \Rightarrow A_2 a \Rightarrow A_1 d a$ , we list in Table 2.5 the steps producing grammar  $G'_3$ , which has no l-recursion.

**Table 2.5** Turning recursion from left to right.

$i$	$j$	Grammar
1		Eliminate immediate l-recursions of $A_1$ (none)
2	1	Replace $A_2 \rightarrow A_1 d$ with the rules constructed expanding $A_1$ , obtaining:  $A_1 \rightarrow A_2 a \mid b$ $A_2 \rightarrow A_2 c \mid A_2 a d \mid b d \mid e$  Eliminate the immediate l-recursion, obtaining $G'_3$ :  $A_1 \rightarrow A_2 a \mid b$ $A_2 \rightarrow b d A'_2 \mid e A'_2 \mid b d \mid e$ $A'_2 \rightarrow c A'_2 \mid a d A'_2 \mid c \mid a d$

It would be straightforward to modify the algorithm to transform right recursions to left ones, a conversion sometimes applied to speed up the bottom-up parsing algorithms of Chapter 5.

<sup>25</sup> A proof of correctness may be found in [28] or in [14].

## Greibach and Real-Time Normal Form

In the *real-time* normal form every rule starts with a terminal:

$$A \rightarrow a\alpha \quad \text{where } a \in \Sigma, \quad \alpha \in \{\Sigma \cup V\}^*$$

A special case of this is *Greibach* normal form:

$$A \rightarrow a\alpha \quad \text{where } a \in \Sigma, \quad \alpha \in V^*$$

Every rule starts with a terminal, followed by zero or more nonterminals.

To be exact, both forms exclude the empty string from the language.

The designation ‘real time’ will be later understood, as a property of the parsing algorithm: at each step it reads and consumes a terminal character, thus the total number of steps equals the length of the string to be parsed.

Assuming for simplicity the given grammar to be nonnullable, we explain how to proceed to obtain the above forms.

For the real-time form: first eliminate all left-recursions; then, by elementary transformations, expand any nonterminal that occurs in first position in a right part, until a terminal prefix is produced. Then continue for the Greibach form: if in any position other than the first, a terminal occurs, replace it by an ancillary nonterminal and add the terminal rule that derives it.

*Example 2.64.* The grammar

$$A_1 \rightarrow A_2a \quad A_2 \rightarrow A_1c \mid bA_1 \mid d$$

is converted to Greibach form by the following steps.

1. Eliminate l-recursions by the step

$$A_1 \rightarrow A_2a \quad A_2 \rightarrow A_2ac \mid bA_1 \mid d$$

and then

$$A_1 \rightarrow A_2a \quad A_2 \rightarrow bA_1A'_2 \mid dA'_2 \mid d \mid bA_1 \quad A'_2 \rightarrow acA'_2 \mid ac$$

2. Expand the nonterminals in first position until a terminal prefix is produced:

$$A_1 \rightarrow bA_1A'_2a \mid dA'_2a \mid da \mid bA_1a \quad A_2 \rightarrow bA_1A'_2 \mid dA'_2 \mid d \mid bA_1$$

$$A'_2 \rightarrow acA'_2 \mid ac$$

3. Substitute ancillary nonterminals for any terminal in a position other than one:

$$A_1 \rightarrow bA_1A'_2\langle a \rangle \mid dA'_2\langle a \rangle \mid d\langle a \rangle \mid bA_1\langle a \rangle \quad A_2 \rightarrow bA_1A'_2 \mid dA'_2 \mid d \mid bA_1$$

$$A'_2 \rightarrow a\langle c \rangle A'_2 \mid a\langle c \rangle$$

$$\langle a \rangle \rightarrow a \qquad \langle c \rangle \rightarrow c$$

Halting before the last step, the grammar would be in real-time but not in Greibach's form.

Although not all preceding transformations, especially not the Chomsky and Greibach ones, will be used in this book, practicing with them is recommended as an exercise for becoming fluent in grammar design and manipulation, a skill certainly needed in language and compiler engineering.

## 2.6 Grammars of Regular Languages

Since regular languages are a rather limited class of context-free languages, it is not surprising that their grammars admit severe restrictions, to be next considered. Furthering the study of regular languages, we shall also see that longer sentences present unavoidable repetitions, a property that can be exploited to prove that certain context-free languages are not regular. Other contrastive properties of the *REG* and *CF* families will emerge in chapters 3 and 4 from consideration of the amount of memory needed to check whether a string is in the language, which is finite for the former and unbounded for the latter family.

### 2.6.1 From Regular Expressions to Context-Free Grammars

Given an r.e. it is straightforward to write a grammar for the language, by analyzing the expression and mapping its subexpressions into grammar rules. At the heart of the construction, the iterative operators (star and cross) are replaced by unilaterally recursive rules.

*Algorithm.* From r.e. to grammar

First we identify and number the subexpressions contained in the given r.e.  $r$ . From the very definition of r.e., the possible cases and corresponding grammar rules (with uppercase nonterminals) are in Table 2.6. Notice we allow the empty string as term. For shortening the grammar, if in any row a term  $r_i$  is a terminal or  $\varepsilon$ , we do not introduce a corresponding nonterminal  $E_i$ , but write it directly in the rule.

Notice that rows 3 and 4 offer the choice of left or right-recursive rules. To apply this conversion scheme, each subexpression label is assigned as a distinguishing subscript to a nonterminal. The axiom is associated with the

**Table 2.6** From subexpressions to grammar rules.

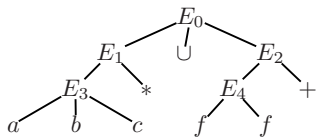
	<i>subexpression</i>	<i>grammar rule</i>
1	$r = r_1.r_2.\dots.r_k$	$E \rightarrow E_1E_2\dots E_k$
2	$r = r_1 \cup r_2 \cup \dots \cup r_k$	$E \rightarrow E_1 \cup E_2 \cup \dots \cup E_k$
3	$r = (r_1)^*$	$E \rightarrow EE_1 \mid \varepsilon$ or $E \rightarrow E_1E \mid \varepsilon$
4	$r = (r_1)^+$	$E \rightarrow EE_1 \mid E_1$ or $E \rightarrow E_1E \mid E_1$
5	$r = b \in \Sigma$	$E \rightarrow b$
6	$r = \varepsilon$	$E \rightarrow \varepsilon$

first step, i.e., to the whole r.e. An example should suffice to understand the procedure.

*Example 2.65.* From r.e. to grammar  
The expression

$$E = (abc)^* \cup (ff)^+$$

is analyzed into the arbitrarily numbered subexpressions shown in the tree, which is a sort of syntax tree of the r.e. with added numbering:



We see that  $E_0$  is union of subexpressions  $E_1$  and  $E_2$ ,  $E_1$  is star of subexpression  $E_3$ , etc.

The mapping scheme of Table 2.6 yields the rules in Table 2.7. The axiom derives the sentential forms  $E_1$  and  $E_2$ ; nonterminal  $E_1$  generates the string forms  $E_3^*$ , and from them  $(abc)^*$ . Similarly  $E_2$  generates strings  $E_4^+$  and finally  $(ff)^+$ .

**Table 2.7** Mapping the r.e. of example 2.65 on grammar rules.

Mapping	Subexpression	Grammar rule
2	$E_1 \cup E_2$	$E_0 \rightarrow E_1 \mid E_2$
3	$E_3^*$	$E_1 \rightarrow E_1 E_3 \mid \varepsilon$
4	$E_4^+$	$E_2 \rightarrow E_2 E_4 \mid E_4$
1	$a b c$	$E_3 \rightarrow a b c$
1	$f f$	$E_4 \rightarrow f f$

Notice that if the r.e. is ambiguous (p. 22), the grammar is so (see example 2.68 on p. 70).

We have thus seen how to map each operator of an r.e. on equivalent rules, to generate the same language. It follows that every regular language



is context-free. Since we know of context-free languages which are not regular (e.g., palindromes and Dyck language), the following property holds.

*Property 2.66.* The family of regular languages  $REG$  is strictly included in the family of context-free languages  $CF$ , that is,  $REG \subset CF$ .

### 2.6.2 Linear Grammars

Algorithm 2.6.1 converts an r.e. to a grammar substantially preserving the structure of the r.e. But for a regular language it is possible to constrain the grammar to a very simple form of rules, called unilinear or of type 3. Such form gives evidence to some fundamental properties and leads to a straightforward construction of the automaton which recognizes the strings of a regular language.

We recall a grammar is *linear* if every rule has the form

$$A \rightarrow uBv \quad \text{where } u, v \in \Sigma^*, B \in (V \cup \epsilon)$$

i.e., at most one nonterminal is in the right part.

Visualizing a corresponding syntax tree, we see it never branches into two subtrees but it has a linear structure made by a stem with leaves directly attached to it. Linear grammars are not powerful enough to generate all context-free languages (an example is Dyck language), but already exceed the power needed for regular languages. For instance, the following well-known subset of Dyck language is generated by a linear grammar but is not regular (to be proved on p. 77).

*Example 2.67.* Nonregular linear language

$$L_1 = \{b^n e^n \mid n \geq 1\} = \{be, bbee, \dots\}$$

Linear grammar:  $S \rightarrow bSe \mid be$

A rule of the following form is called *right-linear*:

$$A \rightarrow uB \quad \text{where } u \in \Sigma^*, B \in (V \cup \epsilon)$$

Symmetrically, a *left-linear* rule has the form

$$A \rightarrow Bu, \text{ with the same stipulations.}$$

Clearly both cases are linear and are obtained by deleting on either side a terminal string embracing nonterminal  $B$ .

A grammar such that all the rules are right-linear or all the rules are left-linear is termed *unilinear* or of *type 3*.<sup>26</sup>

---

<sup>26</sup> Within Chomsky hierarchy (p. 87).

For a right-linear grammar every syntax tree has an oblique stem oriented towards the right (towards the left for a left-linear grammar). Moreover, if the grammar is recursive, it is necessarily right-recursive.

*Example 2.68.* The strings containing  $aa$  and ending with  $b$  are defined by the (ambiguous) r.e.

$$(a \mid b)^*aa(a \mid b)^*b$$

The language is generated by the unilinear grammars:

1. Right-linear grammar  $G_r$ :

$$S \rightarrow aS \mid bS \mid aaA \quad A \rightarrow aA \mid bA \mid b$$

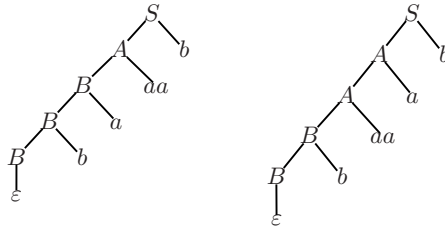
2. Left-linear grammar  $G_l$ :

$$S \rightarrow Ab \quad A \rightarrow Aa \mid Ab \mid Baa \quad B \rightarrow Ba \mid Bb \mid \varepsilon$$

An equivalent nonunilinear grammar is constructed by the algorithm on p. 67:

$$E_1 \rightarrow E_2aaE_2b \quad E_2 \rightarrow E_2a \mid E_2b \mid \varepsilon$$

With grammar  $G_l$  the leftwards syntax trees of the ambiguous sentence  $baaab$  are



*Example 2.69.* Parenthesis-free arithmetic expressions

The language

$$L = \{a, a + a, a * a, a + a * a, \dots\}$$

is defined by the right-linear grammar  $G_r$ :

$$S \rightarrow a \mid a + S \mid a * S$$

or by the left-linear grammar  $G_l$ :

$$S \rightarrow a \mid S + a \mid S * a$$

By the way, neither grammar is adequate to impose the precedence of arithmetic operations.

### Strictly Unilinear Grammars

The unilinear rule form can be further constrained, with the aim of simplifying the coming discussion of theoretical properties and the construction of language recognizing automata. A grammar is *strictly unilinear* if every rule contains at most one terminal character, i.e., if it has the form

$$A \rightarrow aB \quad (\text{or } A \rightarrow Ba), \text{ where } a \in (\Sigma \cup \varepsilon), B \in (V \cup \varepsilon)$$

A further simplification is possible: to impose that the only terminal rules are empty ones. In this case we may assume the grammar contains just the following rule types:

$$A \rightarrow aB \mid \varepsilon \quad \text{where } a \in \Sigma, B \in V$$

Summarizing the discussion, we may indifferently use a grammar in unilinear form or strictly unilinear form, and we may additionally choose to have as terminal rules only the empty ones.

*Example 2.70.* Example 2.69 continued

By adding ancillary nonterminals, the right-linear grammar  $G_r$  is transformed to the equivalent strictly right-linear grammar  $G'_r$ :

$$S \rightarrow a \mid aA \quad A \rightarrow +S \mid *S$$

and also to the equivalent grammar with null terminal rules:

$$S \rightarrow aA \quad A \rightarrow +S \mid *S \mid \varepsilon$$

### 2.6.3 Linear Language Equations

Continuing the study of unilinear grammars, we show the languages they generate are regular. The proof consists of transforming the rules to a set of linear equations, having regular languages as their solution. In Chapter 3 we shall see that every regular language can be defined by a unilinear grammar, thus proving the identity of the languages defined by r.e. and by unilinear grammars.

For simplicity take a grammar  $G = (V, \Sigma, P, S)$  in strictly right-linear form (the case of left-linear grammar is analogous) with null terminal rules. Any such rule can be transcribed into a linear equation having as unknowns the languages generated from each nonterminal, that is, for nonterminal  $A$

$$L_A = \{x \in \Sigma^* \mid A \stackrel{+}{\Rightarrow} x\}$$

and in particular,  $L(G) \equiv L_S$ .

A string  $x \in \Sigma^*$  is in language  $L_A$  if:

- $x$  is the empty string and  $P$  contains rule  $A \rightarrow \varepsilon$ ;
- $x$  is the empty string,  $P$  contains rule  $A \rightarrow B$  and  $\varepsilon \in L_B$ ;
- $x = ay$  starts with character  $a$ ,  $P$  contains rule  $A \rightarrow aB$  and string  $y \in \Sigma^*$  is in language  $L_B$ .

Let  $n = |V|$  be the number of nonterminals. Each nonterminal  $A_i$  is defined by a set of alternatives

$$A_i \rightarrow aA_1 \mid bA_1 \mid \dots \mid \dots \mid aA_n \mid bA_n \mid \dots \mid A_1 \mid \dots \mid A_n \mid \varepsilon$$

some possibly missing.<sup>27</sup> We write the corresponding equation:

$$L_{A_i} = aL_{A_1} \cup bL_{A_1} \cup \dots \cup aL_{A_n} \cup bL_{A_n} \cup \dots \cup L_{A_1} \cup \dots \cup L_{A_n} \cup \varepsilon$$

The last term disappears if the rule does not contain the alternative  $A_i \rightarrow \varepsilon$ . This system of  $n$  simultaneous equations in  $n$  unknowns (the languages generated by the nonterminals) can be solved by the well-known method of Gaussian elimination, by applying the following formula to break recursion.

*Property 2.71.* Arden identity

The equation

$$X = KX \cup L \tag{2.7}$$

where  $K$  is a nonempty language and  $L$  any language, has one and only one solution

$$X = K^*L \tag{2.8}$$

It is simple to see language  $K^*L$  is a solution of (2.7) because, substituting it for the unknown in both sides, the equation turns into the identity

$$K^*L = (KK^*L) \cup L$$

It would also be possible to prove that equation (2.7) has no solution other than (2.8).

*Example 2.72.* Language equations

The right-linear grammar

$$S \rightarrow sS \mid eA \quad A \rightarrow sS \mid \varepsilon$$

defines a list of (possibly missing) elements  $e$ , divided by separator  $s$ . It is transcribed to the system

$$\begin{cases} L_S &= sL_S \cup eL_A \\ L_A &= sL_S \cup \varepsilon \end{cases}$$

Substitute the second equation into the first:

---

<sup>27</sup> In particular, alternative  $A_i \rightarrow A_i$  is never present since the grammar is noncircular.

$$\begin{cases} L_S &= sL_S \cup e(sL_S \cup \varepsilon) \\ L_A &= sL_S \cup \varepsilon \end{cases}$$

Then apply the distributive property of concatenation over union, to factorize variable  $L_S$  as a common suffix:

$$\begin{cases} L_S &= (s \cup es)L_S \cup e \\ L_A &= sL_S \cup \varepsilon \end{cases}$$

Apply Arden identity to the first equation, obtaining

$$\begin{cases} L_S &= (s \cup es)^*e \\ L_A &= sL_S \cup \varepsilon \end{cases}$$

and then  $L_A = s(s \cup es)^*e \cup \varepsilon$ .

Notice that it is straightforward to write the equations also for unilinear grammars, which are not strictly so. We have thus proved that every unilinearly generated language is regular.

An alternative method for computing the r.e. of a language defined by a finite automaton will be described in Chapter 3.

## 2.7 Comparison of Regular and Context-Free Languages

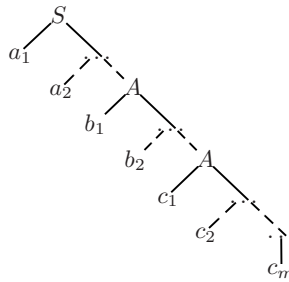
It is important to understand the scope of regular languages, in order to realize which constructs can be thus defined, and which require the full power of context-free grammars. To this end we present a structural property of regular languages. Recall first that, in order to generate an infinite language, a grammar must be recursive (property 2.37, p. 39), because only recursive derivations, such as  $A \xRightarrow{+} uAv$ , can be iterated  $n$  (unbounded) times, producing the string  $u^nAv^n$ . This fact leads to the observation that any sufficiently large sentence necessarily includes a recursive derivation in its generation; therefore it contains certain substrings that can be unboundedly repeated, producing a longer sentence of the language.

This observation will be stated more precisely, first for unilateral, then for context-free grammars.

*Property 2.73.* Pumping of strings

Take a unilinear grammar  $G$ . For any sufficiently long sentence  $x$ , meaning of length greater than some grammar dependent constant, it is possible to find a factorization  $x = tuv$ , where string  $u$  is not empty, such that, for every  $n \geq 0$ , the string  $tu^n v$  is in the language. (It is customary to say the given sentence can be “pumped” by injecting arbitrarily many times the substring  $u$ .)

Proof. Take a strictly right-linear grammar and let  $k$  be the number of non-terminal symbols. Observe the syntax tree of any sentence  $x$  of length  $k$  or more; clearly two nodes exist with the same nonterminal label  $A$ :



Consider the factorization into  $t = a_1a_2\dots$ ,  $u = b_1b_2\dots$ , and  $v = c_1c_2\dots c_m$ . Therefore there is a recursive derivation:

$$S \stackrel{+}{\Rightarrow} tA \stackrel{+}{\Rightarrow} tuA \stackrel{+}{\Rightarrow} tuv$$

that can be repeated to generate the strings  $tuuv$ ,  $tu\dots uv$  and  $tv$ .

This property is next exploited to demonstrate that a language is not regular.

*Example 2.74.* Language with two equal powers

Consider the familiar context-free language

$$L_1 = \{b^n e^n \mid n \geq 1\}$$

and assume by contradiction it is regular. Take a sentence  $x = b^k e^k$ , with  $k$  large enough, and decompose it into three substrings,  $x = tuv$ , with  $u$  not empty. Depending on the positions of the two divisions, the strings  $t$ ,  $u$ , and  $v$  are as in the following scheme:

$$\begin{array}{c} \underbrace{b\dots b}_t \underbrace{b\dots b}_u \underbrace{b\dots be\dots e}_v \\ \underbrace{b\dots b}_t \underbrace{b\dots be\dots e}_u \underbrace{e\dots e}_v \\ \underbrace{b\dots be\dots e}_t \underbrace{e\dots e}_u \underbrace{e\dots e}_v \end{array}$$

Pumping the middle string will lead to contradiction in all cases. For row one, if  $u$  is repeated twice, the number of  $b$  exceeds the number of  $e$ , causing the pumped string not to be in the language. For row two, repeating twice  $u$ , the string  $tuuv$  contains a pair of substrings  $be$  and does not conform to the language structure. Finally for row three, repeating  $u$ , the number of  $e$  exceeds the number of  $b$ . In all cases the pumped strings are not valid and property 2.73 is contradicted. This completes the proof that the language is not regular.

This example and the known inclusion of the families  $REG \subseteq CF$  justify the following statement.

*Property 2.75.* Every regular language is context-free and there exist context-free languages which are not regular.

The reader should be convinced by this example, that the regular family is too narrow for modelling some typical simple constructs of technical languages. Yet it would be foolish to discard regular expressions, because they are perfectly fit for modelling some most common parts of technical languages: on one hand there are the substrings that make the so-called lexicon (for instance, numerical constants and identifiers), on the other hand, many constructs that are variations over the list paradigm (e.g., lists of procedure parameters or of instructions).

### Role of Self-Nested Derivations

Having ascertained that regular languages are a smaller family than context-free ones, it is interesting to focus on what makes some typical languages (as the two powers language, Dyck or palindromes) not regular. Careful observation reveals that their grammars have a common feature: they all use some recursive derivation which is neither left nor right, but is called *self-nested*:

$$A \stackrel{\pm}{\Rightarrow} uAv \quad u \neq \varepsilon \text{ and } v \neq \varepsilon$$

On the contrary, such derivations cannot be obtained with unilinear grammars which permit only unilateral recursions.

Now, it is the absence of self-nesting recursion that permitted us to solve linear equations by Arden identity. The higher generative capacity of context-free grammars essentially comes from such derivations, as next stated.

*Property 2.76.* Any context-free grammar not producing self-nesting derivations generates a regular language.

*Example 2.77.* Not self-nesting grammar

The grammar  $G$ :

$$S \rightarrow AS \mid bA \quad A \rightarrow aA \mid \varepsilon$$

though not unilinear, does not permit self-nested derivations. Therefore  $L(G)$  is regular, as we can see by solving the language equations.

$$\begin{aligned} \begin{cases} L_S &= L_A L_S \cup bL_A \\ L_A &= aL_A \cup \varepsilon \end{cases} \\ \begin{cases} L_S &= L_A L_S \cup bL_A \\ L_A &= a^* \end{cases} \\ L_S &= a^* L_S \cup ba^* \end{aligned}$$

$$L_S = (a^*)^*ba^*$$

### Context-Free Languages of Unary Alphabet

The converse of property 2.76 is not true in general: in some cases self-nesting derivations do not cause the language to be nonregular. On the way to illustrate this fact, we take the opportunity to mention a curious property of context-free languages having a one-letter alphabet.

*Property 2.78.* Every language defined by a context-free grammar over a one-letter (or unary) alphabet,  $|\Sigma| = 1$ , is regular.

Observe that the sentences  $x$  with unary alphabet are in bijective correspondence with integer numbers, via the mapping  $x \Leftrightarrow n$ , if and only if  $|x| = n$ .

*Example 2.79.* The grammar

$$G = \{S \rightarrow aSa \mid \varepsilon\}$$

has the self-nesting derivation  $S \Rightarrow aSa$ , but  $L(G) = (aa)^*$  is regular. A right-linear, equivalent grammar is easily obtained, by shifting to suffix the nonterminal that is, positioned in the middle of the first rule:

$$\{S \rightarrow aaS \mid \varepsilon\}$$

### 2.7.1 Limits of Context-Free Languages

In order to understand what cannot be done with context-free grammars, we study the unavoidable repetitions which are found in longer sentences of such languages, much as we did for regular languages. We shall see that longer sentences necessarily contain two substrings, which can be repeated the same unbounded number of times by applying a self-nested derivation. This property will be exploited to prove that context-free grammars cannot generate certain languages where three or more parts are repeated the same number of times.

*Property 2.80.* Language with three equal powers  
The language

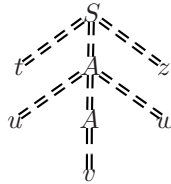
$$L = \{a^n b^n c^n \mid n \geq 1\}$$

is not context-free.

*Proof.* By contradiction, assume grammar  $G$  of  $L$  exists and imagine the syntax tree of sentence  $x = a^n b^n c^n$ . Focus now on the paths from the root (axiom  $S$ ) to the leaves. At least one path must have a length that increases with the length of sentence  $x$ , and since  $n$  can be arbitrarily large, such path



necessarily traverses two nodes with identical nonterminal label, say  $A$ . The situation is depicted in the scheme:



where  $t, u, v, w, z$  are terminal strings. This scheme denotes the derivation

$$S \Rightarrow^+ tAz \Rightarrow^+ tuAwz \Rightarrow^+ tuv wz$$

This contains a recursive subderivation from  $A$  to  $A$ , which can be repeated any number  $j$  of times, producing strings of type

$$y = t \overbrace{u \dots u}^j v \overbrace{w \dots w}^j z$$

Now, examine all possible cases for strings  $u, w$ :

- Both strings contain just one and the same character, say  $a$ ; therefore, as  $j$  increases, string  $y$  will have more  $a$  than  $b$ , hence cannot be in the language.
- String  $u$  contains two or more different characters, for instance  $u = \dots a \dots b \dots$ . Then, by repeating the recursive part of the derivation, we obtain  $uu = \dots a \dots b \dots a \dots b \dots$ , where characters  $a$  and  $b$  are mixed up, hence  $y$  is not in the language.  
We do not discuss the analogous case when string  $w$  contains two different characters.
- String  $u$  contains only one character, say  $a$ , and string  $w$  only one different character, say  $b$ . When  $j$  increases, string  $y$  contains a number of  $a$  greater than the number of  $c$ , hence it is not valid.

This reasoning exploits the possibility of pumping the sentences by repeating a recursive derivation. It is a useful conceptual tool for proving that certain languages are not in the  $CF$  family.

Although the language with three equal powers has no practical relevance, it illustrates a kind of agreement or concordance that cannot be enforced by context-free rules. The next case considers a construct more relevant for technical languages.

### Language of Copies or Replica

An outstanding abstract paradigm is the *replica*, to be found in many technical contexts, whenever two lists contain elements that must be identical

or more generally must agree with each other. A concrete case is provided by procedure declaration/invoke: the correspondence between the formal parameter list and the actual parameter list. An example inspired by English is: cats, vipers, crickets, and lions are respectively mammals, snakes, insects, and mammals.

In the most abstract form the two lists are made with the same alphabet and the replica is the language

$$L_{\text{replica}} = \{uu \mid u \in \Sigma^+\}$$

Let  $\Sigma = \{a, b\}$ . A sentence  $x = abbbabbb = uu$  is in some respect analogous to a palindrome  $y = abbbbbba = uu^R$ , but string  $u$  is copied in the former language, specularly reversed in the latter. We may say the symmetry of sentences  $L_{\text{replica}}$  is translational, not specular. Strange enough, whereas palindromes are a most simple context-free language, the language of replicas is not context-free. This comes from the fact that the two forms of symmetry require quite different control mechanisms: a LIFO (last in first out) push-down stack for specular, and a FIFO (first in first out) queue for translational symmetry. We shall see in chapter 4 that the algorithms (or automata) recognizing context-free languages use a LIFO memory.

In order to show that replica is not in  $CF$ , one should apply again the pumping reasoning; but before doing so, we have to filter the language to render it similar to the three equal powers language.

We focus on the following subset of  $L_{\text{replica}}$ , obtained by means of intersection with a regular language:

$$L_{abab} = \{a^m b^n a^m b^n \mid m, n \geq 1\} = L_{\text{replica}} \cap a^+ b^+ a^+ b^+$$

We state (anticipating the proof on p. 160) that the intersection of a context-free language with a regular one is always a context-free language. Therefore, if we prove that  $L_{abab}$  is not context-free, we may conclude the same for  $L_{\text{replica}}$ .

For brevity, we omit the analysis of the possible cases of the strings to be pumped, since it closely resembles the discussion in the previous proof (p. 76).

### 2.7.2 Closure Properties of REG and CF

We know language operations are used to combine languages into new ones with the aim of extending, filtering, or modifying a given language. But not all operations preserve the class or family the given languages belong to. When the result of the operation exits from the starting family, it can no longer be generated with the same type of grammar.

Continuing the comparison between regular and context-free languages, we resume in Table 2.8 the closure properties with respect to language operations: some are already known, others are immediate, and a few need to await the results of automata theory for their proofs. We denote as  $L$  and  $R$  a generic context-free language and regular language, respectively.

**Table 2.8** Closure properties of  $REG$  and  $CF$ .

<i>reflection</i>	<i>star</i>	<i>union or concatenation</i>	<i>complement</i>	<i>intersection</i>
$R^R \in REG$	$R^* \in REG$	$R_1 \oplus R_2 \in REG$	$\neg R \in REG$	$R_1 \cap R_2 \in REG$
$L^R \in CF$	$L^* \in CF$	$L_1 \oplus L_2 \in CF$	$\neg L \notin CF$	$L_1 \cap L_2 \notin CF$
				$L \cap R \in CF$

Comments and examples follow.

- A nonmembership (such as  $\neg L \notin CF$ ) means that the left term does not always belong to the family; but this does not exclude, for instance, that the complement of some context-free language is context-free.
- The mirror language of  $L(G)$  is generated by the *mirror grammar*, the one obtained reversing the right parts of the rules. Clearly, if grammar  $G$  is right-linear the mirror grammar is left-linear and defines a regular language.
- We know the star, union, and concatenation of context-free languages are context-free. Let  $G_1$  and  $G_2$  be the grammars of  $L_1$  and  $L_2$ , let  $S_1$  and  $S_2$  be their axioms, and suppose that the nonterminal sets are disjoint,  $V_1 \cap V_2 = \emptyset$ . To obtain the new grammar in the three cases, add to the rules of  $G_1$  and  $G_2$  the following initial rules:

$$\begin{aligned}
 \text{Star:} & \quad S \rightarrow SS_1 \mid \varepsilon \\
 \text{Union:} & \quad S \rightarrow S_1 \mid S_2 \\
 \text{Concatenation:} & \quad S \rightarrow S_1 S_2
 \end{aligned}$$

In the case of union, if the grammars are right-linear, so is the new grammar. On the contrary, the new rules introduced for concatenation and star are not right-linear but we know that an equivalent right-linear grammar for the resulting languages exists, because they are regular (property 2.23, p. 24) and family  $REG$  is closed by such operations.

- The proof that the complement of a regular language is regular is in Chapter 3 (p.137).
- The intersection of two context-free languages is not context-free (in general), as witnessed by the known language with three equal powers (example 2.80 on p. 76)

$$\{a^n b^n c^n \mid n \geq 1\} = \{a^n b^n c^+ \mid n \geq 1\} \cap \{a^+ b^n c^n \mid n \geq 1\}$$

where the two components are easily defined by context-free grammars.

- As a consequence of De Morgan identity, the complement of a context-free language is not context-free (in general): since  $L_1 \cap L_2 = \neg(\neg L_1 \cup \neg L_2)$ , if the complement were context-free, a contradiction would ensue since the union of two context-free languages is context-free.
- On the other hand, the intersection of a context-free and a regular language is context-free. The proof will be given on p. 160.

The last property can be applied, in order to make a grammar more discriminatory, by filtering the language with a regular language which forces some constraints on the original sentences.

*Example 2.81.* Regular filters on Dyck language (p. 45)

It is instructive to see how the freely parenthesized sentences of a Dyck language  $L_D$  of alphabet  $\Sigma = \{a, a'\}$  can be filtered, by intersecting with the regular languages:

$$\begin{aligned} L_1 &= L_D \cap \neg(\Sigma^* a' a' \Sigma^*) = (aa')^* \\ L_2 &= L_D \cap \neg(\Sigma^* a' a \Sigma^*) = \{a^n (a')^n \mid n \geq 0\} \end{aligned}$$

The first intersection preserves the sentences that do not contain substring  $a'a'$ , i.e., it eliminates all the sentences with nested parentheses. The second filter preserves the sentences having exactly one nest of parentheses. Both results are context-free languages, but the former is also regular.

### 2.7.3 Alphabetic Transformations

It is a common experience to find conceptually similar languages that differ by the concrete syntax, i.e., by the choice of terminals. For instance, multiplication may be represented by sign  $\times$ , by an asterisk, or by a dot in different languages.

The term *transliteration* or *alphabetic homomorphism* refers to the linguistic operation that replaces individual characters by other ones.

**Definition 2.82.** Transliteration (or alphabetic homomorphism<sup>28</sup>)

Consider two alphabets: *source*  $\Sigma$  and *target*  $\Delta$ . An alphabetic transliteration is a function:

$$h : \Sigma \rightarrow \Delta \cup \{\varepsilon\}$$

The transliteration or image of character  $c \in \Sigma$  is  $h(c)$ , an element of the target alphabet. If  $h(c) = \varepsilon$ , character  $c$  is erased. A transliteration is *nonerasing* if, for no source character  $c$ , it is  $h(c) = \varepsilon$ .

The image of a source string  $a_1 a_2 \dots a_n$ ,  $a_i \in \Sigma$  is the string  $h(a_1)h(a_2) \dots h(a_n)$  obtained concatenating the images of individual characters. Notice the image of the empty string is itself.

---

<sup>28</sup> This is a simple case of the translation functions to be studied in Chapter 6.

Such transformation is compositional: the image of the concatenation of two strings  $v$  and  $w$  is the concatenation of the images of the strings:

$$h(v.w) = h(v).h(w)$$

*Example 2.83. Printer*

An obsolete printer cannot print Greek characters and instead prints the special character  $\square$ . Moreover, the test sent to the printer may contain control characters (such as **start-text**, **end-text**) that are not printed. The text transformation (disregarding uppercase letters) is described by the transliteration:

$$\begin{aligned} h(c) &= c && \text{if } c \in \{a, b, \dots, z, 0, 1, \dots, 9\}; \\ h(c) &= c && \text{if } c \text{ is a punctuation mark or a blank space;} \\ h(c) &= \square && \text{if } c \in \{\alpha, \beta, \dots, \omega\}; \\ h(\text{start-text}) &= h(\text{end-text}) = \varepsilon. \end{aligned}$$

An example of transliteration is

$$h(\underbrace{\text{start-text the const. } \pi \text{ has value 3.14 end-text}}_{\text{source string}}) = \underbrace{\text{the const. } \square \text{ has value 3.14}}_{\text{target string}}$$

An interesting special case of erasing homomorphism is the *projection*: it is a function that erases some source characters and leaves the others unchanged.

### Transliteration to Words

The preceding qualification of transliteration as alphabetic means the image of a character is a character (or the empty string), not a longer string. Otherwise the transliteration or homomorphism may no longer be qualified as alphabetic. An example is the conversion of an assignment statement  $a \leftarrow b + c$  to the form  $a := b + c$  by means of a (nonalphabetic) transliteration:

$$h(\leftarrow) = ':=' \quad h(c) = c \quad \text{for any other } c \in \Sigma$$

This case is also called transliteration *to words*.

Another example: vowels with umlaut of German alphabet have as image a string of two characters:

$$h(\ddot{a}) = ae, \quad h(\ddot{o}) = oe, \quad h(\ddot{u}) = ue$$

### Language Substitution

A further generalization leads us to a language transformation termed *substitution* (already introduced in the discussion of linguistic abstraction on

p. 27). Now a source character can be replaced by any string of a specified language. Substitution is very useful in early language design phases, in order to leave a construct unspecified in the working grammar. The construct is denoted by a symbol (for instance  $\langle \text{identifier} \rangle$ ). As the project progresses, the symbol will be substituted with the definition of the corresponding language (for instance with  $(a \dots z)(a \dots z \mid 0 \dots 9)^*$ ).

par Formally, given a source alphabet  $\Sigma = \{a, b, \dots\}$ , a substitution  $h$  associates each character with a language  $h(a) = L_a, h(b) = L_b, \dots$  of target alphabet  $\Delta$ . Applying substitution  $h$  to a source string  $a_1 a_2 \dots a_n, a_i \in \Sigma$  we obtain a set of strings:

$$h(a_1 a_2 \dots a_n) = \{y_1 y_2 \dots y_n \mid y_i \in L_{a_i}\}$$

We may say a transliteration to words is a substitution such that each image language contains one string only; if the string has length one or zero, the transliteration is alphabetic.

## Closure under Alphabetic Transformation

Let  $L$  be a source language, context-free or regular, and  $h$  a substitution such that for every source character, its image is a language in the same family as the source language. Then the substitution maps the set of source sentences (i.e.,  $L$ ) on a set of image strings, called the *image* or *target* language,  $L' = h(L)$ . Is the target language a member of the same family as the source language? The answer is yes, and will be given by means of a construction that is, valuable for modifying without effort the regular expression or the grammar of the source language.

*Property 2.84.* The family  $CF$  is closed by the operation of substitution with languages of the same family (therefore also by the operation of transliteration).

*Proof.* Let  $G$  be the grammar of  $L$  and  $h$  a substitution such that, for every  $c \in \Sigma$ ,  $L_c$  is context-free. Let this language be defined by grammar  $G_c$  with axiom  $S_c$ . Moreover, we assume the nonterminal sets of grammars  $G, G_a, G_b, \dots$  are pairwise disjoint (otherwise it suffices to rename the overlapping nonterminals).

Next we construct the grammar  $G'$  of language  $h(L)$  by transliterating the rules of  $G$  with the following mapping  $f$ :

$$\begin{aligned} f(c) &= S_c, \text{ for every terminal } c \in \Sigma; \\ f(A) &= A, \text{ for every nonterminal symbol } A \text{ of } G. \end{aligned}$$

The rules of grammar  $G'$  are constructed next:

- to every rule  $A \rightarrow \alpha$  of  $G$  apply transliteration  $f$ , to the effect of replacing each terminal character with the axiom of the corresponding target grammar;

- add the rules of grammars  $G_a, G_b, \dots$

It should be clear that the new grammar generates language  $h(L(G))$ .

In the simple case where the substitution  $h$  is a transliteration, the construction of grammar  $G'$  is more direct: replace in  $G$  any terminal character  $c \in \Sigma$  with its image  $h(c)$ .

For regular languages an analogous result holds.

*Property 2.85.* The family *REG* is closed by substitution (therefore also by transliteration) with regular languages.

Essentially the same construction of the proof of property 2.84 can be applied to the r.e. of the source language, to compute the r.e. of the target language.

*Example 2.86.* Grammar transliterated

The source language  $i(;i)^*$ , defined by rules

$$S \rightarrow i; S \mid i$$

schematizes a program including a list of instructions  $i$  separated by semicolon. Imagine that instructions have to be now defined as assignments. Then the following transliteration to words is appropriate:

$$g(i) = v \leftarrow e$$

where  $v$  is a variable and  $e$  an arithmetic expression. This produces the grammar

$$S \rightarrow A; S \mid A \quad A \rightarrow v \leftarrow e$$

As next refinement, the definition of expressions can be plugged in by means of a substitution  $h(e) = L_E$ , where the image language is the well-known one. Suppose it is defined by a grammar with axiom  $E$ . The grammar of the language after expression expansion is

$$S \rightarrow A; S \mid A \quad A \rightarrow v \leftarrow E \quad E \rightarrow \dots \quad - - \text{usual rules for arith. expr.}$$

As a last refinement, symbol  $v$ , which stands for a variable, should be replaced with the regular language of identifier names.

### 2.7.4 Grammars with Regular Expressions

The legibility of r.e. is especially good for lists and similar structures, and it would be a pity to do without them when defining technical languages by means of grammars. Since we know recursive rules are indispensable for parenthesis structures, the idea arises to combine r.e. and grammar rules in

a notation, called *extended context-free grammar*, or *EBNF*<sup>29</sup> that takes the best of each formalism: simply enough we allow the right part of a rule to be an r.e. Such grammars have a nice graphical representation, the syntax charts to be shown in Chapter 4 on p. 174, which represents the blueprint of the flowchart of the syntax analyzer.

First observe that, since family *CF* is closed by all the operations of r.e., the family of languages defined by *EBNF* grammars coincide with family *CF*.

In order to appreciate the clarity of extended rules with respect to basic ones, we examine a few typical constructs of programming languages.

*Example 2.87.* *EBNF* grammar of a block language: declarations  
Consider a list of variable declarations:

*char* text1, text2; *real* temp, result; *int* alpha, beta2, gamma;

to be found with syntactic variations in most programming languages.

The alphabet is  $\Sigma = \{c, i, r, v, ', ', ', '\}$ , where *c, i, r* stand for *char, int, real* and *v* for a variable name. The language of lists of declarations is defined by r.e. *D*:

$$((c \mid i \mid r)v(, v)^*; )^+$$

The iteration operators used to generate lists are dispensable: remember any regular language can be defined by a grammar, even a unilinear one.

The lists can be defined by the basic grammar

$$D \rightarrow DE \mid E \quad E \rightarrow AN; \quad A \rightarrow c \mid i \mid r \quad N \rightarrow v, N \mid v$$

with two recursive rules (for *D* and *N*). The grammar is a bit longer than the r.e. and subjectively less perspicuous in giving evidence to the two-level hierarchical structure of declarations, which is evident in the r.e. Moreover, the choice of metasymbols *A, E, N* is to some extent mnemonic but arbitrary and may cause confusion, when several individuals jointly design a grammar.

**Definition 2.88.** An *extended context-free* or *EBNF* grammar  $G = (V, \Sigma, P, S)$  contains exactly  $|V|$  rules, each one in the form  $A \rightarrow \alpha$ , where *A* is a non-terminal and  $\alpha$  is an r.e. of alphabet  $V \cup \Sigma$ .

For better legibility and concision, other derived operators (cross, power, option) too may be permitted in the r.e. .

We continue the preceding example, adding typical block structures to the language.

*Example 2.89.* Algol-like language

A block *B* embraces an optional declarative part *D* followed by the imperative part *I*, between the marks *b* (begin) and *e* (end):

---

<sup>29</sup> Extended BNF.



$$B \rightarrow b [D] I e$$

The declarative part  $D$  is taken from the preceding example:

$$D \rightarrow ((c \mid i \mid r)v(v)^*;)^+$$

The imperative part  $I$  is a list of phrases  $F$  separated by semicolon:

$$I \rightarrow F(; F)^*$$

Last, a phrase  $F$  can be an assignment  $a$  or a block  $B$ :

$$F \rightarrow a \mid B$$

As an exercise, yet worsening legibility, we eliminate as many nonterminals as possible by applying nonterminal expansion to  $D$ :

$$B \rightarrow b \left[ ((c \mid i \mid r)v(v)^*;)^+ \right] I e$$

A further expansion of  $I$  leads to

$$B \rightarrow b((c \mid i \mid r)v(v)^*;)^* F(; F)^* e$$

Last  $F$  can be eliminated obtaining a one-rule grammar  $G'$

$$B \rightarrow b((c \mid i \mid r)v(v)^*;)^*(a \mid B)(; (a \mid B))^* e$$

This cannot be reduced to an r.e. because nonterminal  $B$  cannot be eliminated, as it is needed to generate nested blocks  $(bb \dots ee)$  by a self-nesting derivation (in agreement with property 2.76, p. 75).

Usually language reference manuals specify grammars by *EBNF* rules, but beware that excessive grammar conciseness is often contrary to clarity. Moreover, if a grammar is split into smaller rules, it may be easier for the compiler writer to associate simple specific semantic actions to each rule, as we shall see in Chapter 5.

## Derivations and Trees in Extended Context-Free Grammars

The right part  $\alpha$  of an extended rule  $A \rightarrow \alpha$  of grammar  $G$  is an r.e., which in general derives an infinite set of strings: each one can be viewed as the right part of a nonextended rule having unboundedly many alternatives.

For instance,  $A \rightarrow (aB)^+$  stands for a set of rules

$$A \rightarrow aB \mid aBaB \mid \dots$$

The notion of derivation can be defined for extended grammars too, via the notion of derivation for r.e. introduced on p. 20.

Shortly, for an *EBNF* grammar  $G$  consider a rule  $A \rightarrow \alpha$ , where  $\alpha$  is an r.e. possibly containing choice operators (star, cross, union, and option); let  $\alpha'$  be a string deriving from  $\alpha$  (using the definition of derivation of r.e.), not containing any choice operator. For any (possibly empty) strings  $\delta$  and  $\eta$  there is a one-step derivation:

$$\delta A \eta \xRightarrow[G]{} \delta \alpha' \eta$$

Then one can define multi-step derivations starting from the axiom and producing terminal strings, and finally the language generated by an *EBNF* grammar, in the same manner as for basic grammars. Exemplification should be enough.

*Example 2.90.* Derivation for extended grammar of expressions

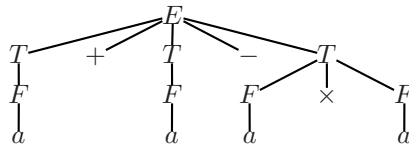
The grammar  $G$ :

$$E \rightarrow [+ \mid -]T((+ \mid -)T)^* \quad T \rightarrow F((\times \mid /)F)^* \quad F \rightarrow (a \mid ' (E')')$$

generates arithmetic expressions with the four infix operators, and the prefix operators  $\pm$ , parentheses, and terminal  $a$  standing for a numeric argument. Square brackets denote option. The left derivation

$$\begin{aligned} E &\Rightarrow T + T - T \Rightarrow F + T - T \Rightarrow a + T - T \Rightarrow a + F - T \Rightarrow a + a - T \Rightarrow \\ &\Rightarrow a + a - F \times F \Rightarrow a + a - a \times F \Rightarrow a + a - a \times a \end{aligned}$$

produces the syntax tree:



Observe that the degree of a node can be unbounded, with the consequence that the breadth of the tree increases and the height decreases, in comparison with the tree of an equivalent basic grammar.

### Ambiguity in Extended Grammars

It is obvious that an ambiguous basic grammar remain such also when viewed as an *EBNF* grammar. On the other hand, a different form of ambiguity may arise in an *EBNF* grammar, caused by the ambiguity of the r.e. present in the rules. Recall an r.e. is ambiguous (p. 22) if it derives a string with two different left derivations.

For instance, the r.e.  $a^*b \mid ab^*$ , numbered  $a_1^*b_2 \mid a_3b_4^*$ , is ambiguous because the string  $ab$  can be derived as  $a_1b_2$  or as  $a_3b_4$ . As a consequence the extended grammar

$$S \rightarrow a^*bS \mid ab^*S \mid c$$

is ambiguous.

## 2.8 More General Grammars and Language Families

We have seen context-free grammars cover the main constructs occurring in technical languages, namely, parentheses structures and hierarchical lists, but fail with other syntactic structures even simple, such as the replica or the three equal powers language on p. 76.

Such shortcomings have motivated, from the early days of language theory, much research on more powerful formal grammars. It is fair to say that none of the formal models have been successful; the more powerful are too obscure and difficult to use, and the models marginally superior to the context-free do not offer significant advantages. In practice, application of such grammars to compilation has been episodic and quickly abandoned. Since the basis of all subsequent developments is the classification of grammars due to the linguist Noam Chomsky, it is appropriate to briefly present it for reference, before moving on to more applied aspects in the coming chapters.

### 2.8.1 Chomsky Classification

The historical classification of phrase structure grammars based on the form of the rewriting rules is in Table 2.9. Surprisingly enough, very small difference in the rule form, determine substantial changes in the properties of the corresponding family of languages, both in terms of decidability and algorithmic complexity.

A rewriting rule has a left part and a right part, both strings on the terminal alphabet  $\Sigma$  and nonterminal set  $V$ . The left part is replaced by the right part. The four types are characterized as follows:

- a rule of *type 0* can replace an arbitrary not empty string over terminals and nonterminals, with another arbitrary string;
- a rule of *type 1* adds a constraint to the form of type 0: the right part of a rule must be at least as long as the left part;
- a rule of *type 2* is context-free: the left part is one nonterminal;
- a rule of *type 3* coincides with the unilinear form we have studied.

For completeness Table 2.9 lists the names of the automata (abstract string recognition algorithms) corresponding to each type, although the notion of automata will not be introduced until next chapter. The language families

**Table 2.9** Chomsky classification of grammars and corresponding languages and machines.

<i>grammar</i>	<i>form of rules</i>	<i>language family</i>	<i>type of recognizer</i>
<i>Type 0</i>	$\beta \rightarrow \alpha$ where $\alpha, \beta \in (\Sigma \cup V)^+$ and $\beta \neq \varepsilon$	Recursively enumerable	Turing machine
<i>Type 1</i> context dependent (or context sensitive)	$\beta \rightarrow \alpha$ where $\alpha, \beta \in (\Sigma \cup V)^+$ and $ \beta  \leq  \alpha $	Contextual or context-dependent	Turing machine with space complexity limited by the length of the source string
<i>Type 2</i> context-free or BNF	$A \rightarrow \alpha$ where $A$ is a nonterminal and $\alpha \in (\Sigma \cup V)^*$	Context-free <i>CF</i> or algebraic	Push-down automaton
<i>Type 3</i> uni-linear (right-linear or left-linear)	Right-linear: $A \rightarrow uB$ Left linear: $A \rightarrow Bu$ , where $A$ is a nonterminal, $u \in \Sigma^*$ and $B \in (V \cup \varepsilon)$	Regular <i>REG</i> or rational or finite-state	Finite automaton

are strictly included one into the next from bottom to top, which justifies the name of hierarchy.

Partly anticipating later matters, the difference between rule types is mirrored by differences in the computational resources needed to recognize the strings. Concerning space, i.e., memory complexity for string recognition, type 3 uses a finite memory, the others need unbounded memory.

Other properties are worth mentioning, without any claim to completeness. All four language families are closed by union, concatenation, star, reflection, and intersection with a regular language. But for other operators, properties differ: for instance families 1 and 3, but not 0 and 2, are closed by complement.

Concerning decidability of various properties, the difference between the apparently similar types 0 and 1 is striking. For type 0 it is undecidable (more precisely semi-decidable) whether a string is in the language generated by a grammar. For type 1 grammars the problem is decidable, though its time complexity is not polynomial. Last, only for type 3 the equivalence problem for two grammars is decidable.

We finish with two examples of type 1 grammars and languages.

*Example 2.91.* Type 1 grammar of the three equal powers language  
The language, proved on p. 76 to be not *CF*, is

$$L = \{a^n b^n c^n \mid n \geq l\}$$

It is generated by the context-sensitive grammar

1.  $S \rightarrow aSBC$

2.  $S \rightarrow abC$

3.  $CB \rightarrow BC$

4.  $bB \rightarrow bb$

5.  $bC \rightarrow bc$

6.  $cC \rightarrow cc$

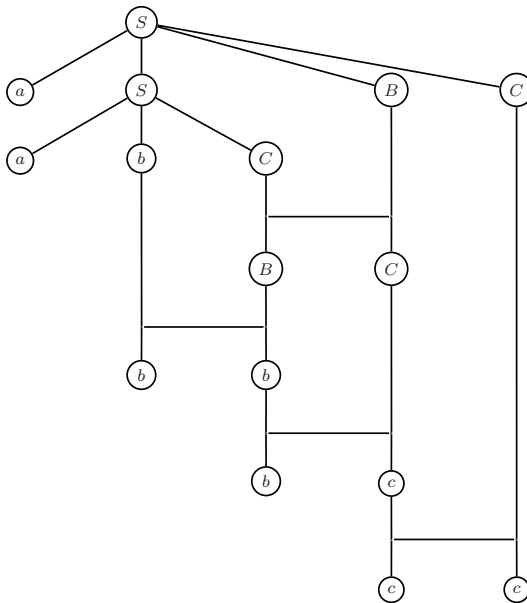
For type 0 and 1 grammars a derivation cannot be represented as a tree, because the left part of a rule typically contains more than one symbol.

However, the derivation can be visualized as a graph, where the application of a rule such as  $BA \rightarrow AB$  is displayed by a bundle of arcs (hyper-edge) connecting the left part nodes to the right part ones.

Coming from our experience with context-free grammars, we would expect to be able to generate all sentences by left derivations. But if we try to generate sentence  $aabbcc$  proceeding from left to right

$$S \Rightarrow aSBC \Rightarrow aabCBC \Rightarrow aabcBC \Rightarrow \text{block!}$$

surprisingly, the derivation is stuck, before all nonterminal symbols are eliminated. In order to generate this string we need a not leftmost derivation, shown in Figure 2.2.



**Fig. 2.2** Graph representation of a context-sensitive derivation (example 2.91).

Intuitively the derivation produces the requested number of  $a$  with rule 1 (a type 2 self-nesting rule) and then with rule 2. In the sentential form, letters  $b$ ,  $B$ , and  $C$  appear with the correct number of occurrences, but mixed up. In order to produce the valid string, all  $C$ 's must be shifted to the right, using rule 3.  $CB \rightarrow BC$ , until they reach the suffix position.

The first application of 3. reorders the sentential form, to the aim that  $B$  becomes adjacent to  $b$ . This enables derivation 4.  $bB \Rightarrow bb$ . Then the derivation continues in the same manner, alternating steps 3. and 4., until the sentential form is left with just  $C$  as nonterminal. The occurrences of

$C$  are transformed to terminals  $c$ , by means of rule  $bC \rightarrow bc$ , followed by repeated applications of rule  $cC \rightarrow cc$ .

We stress the finding that the language generated by type 1 grammars may not coincide with the sentences generated by left derivations, unlike for type 2 grammars. This is a cause of difficulty in string recognition algorithms.

Type 1 grammars have the power to generate the language of replicas, or lists with agreements between elements, a construct that we have singled out as practically relevant and exceeding the power of context-free grammars.

*Example 2.92.* Type 1 grammar of replica with center

Language  $L = \{ycy \mid y \in \{a, b\}^+\}$  contains sentences such as  $aabcaab$ , where a prefix and a suffix, divided by the central separator  $c$ , must be equal.

To simplify the grammar, we assume sentences are terminated to the right by the end-of-text character or *terminator*,  $\vdash$ . Grammar:

$$\begin{array}{lllll}
 S \rightarrow X \vdash & XA \rightarrow XA' & A'A \rightarrow AA' & A' \vdash a & B'a \rightarrow ba \\
 X \rightarrow aXA & XB \rightarrow XB' & A'B \rightarrow BA' & B' \vdash b & B'b \rightarrow bb \\
 X \rightarrow bXB & & B'A \rightarrow AB' & A'a \rightarrow aa & Xa \rightarrow ca \\
 & & B'B \rightarrow BB' & A'b \rightarrow ab & Xb \rightarrow cb
 \end{array}$$

To generate a sentence, the grammar follows this strategy: it first generates a palindrome, say  $aabXBAA$ , where  $X$  marks the center of the sentence and the second half is in uppercase. Then the second half, modified as  $B'AA$ , is reflected and converted in several steps to  $A'A'B'$ . Last, the primed uppercase symbols are rewritten as  $aab$  and the center symbol  $X$  is converted to  $c$ .

We illustrate the derivation of sentence  $aabcaab$ . For legibility, at each step we underline the left part of the rule being applied:

$$\begin{aligned}
 \underline{S} &\Rightarrow \underline{X} \vdash \Rightarrow a\underline{XA} \vdash \Rightarrow aa\underline{XAA} \vdash \Rightarrow aab\underline{XBA}A \vdash \Rightarrow \\
 &aabX\underline{B'AA} \vdash \Rightarrow aab\underline{XAB'}A \vdash \Rightarrow aabXA'\underline{B'A} \vdash \Rightarrow \\
 &aabXA'\underline{AB'} \vdash \Rightarrow aab\underline{XAA'}B' \vdash \Rightarrow aabXA'A'\underline{B'} \vdash \Rightarrow \\
 &aabXA'A'\underline{b} \Rightarrow aabX\underline{A'ab} \Rightarrow aab\underline{Xaab} \Rightarrow aabcaab
 \end{aligned}$$

We observe that the same strategy used in generation, if applied in reverse order, would allow to check whether a string is a valid sentence. Starting from the given string, the algorithm should store on a memory tape the strings obtained after each reduction (i.e., the reverse of derivation). Such procedure is essentially the computation of a Turing machine that never goes out of the portion of tape containing the original string, but may overprint its symbols.

Such examples should have persuaded the reader of the difficulty to design and apply context-sensitive rules even for very simple languages. The fact is that interaction of grammar rules is hard to understand and control.

In other words, type 1 and 0 grammars can be viewed as a particular notation for writing algorithms. All sorts of problems could be programmed

in this way, even mathematical ones, by using the very simple mechanism of string rewriting<sup>30</sup>. It is not surprising that using such elementary mechanism as the only data and control structure, makes the algorithm description very entangled.

Undoubtedly the development of language theory towards models of higher computational capacity has mathematical and speculative interests but is almost irrelevant for language engineering and compilation<sup>31</sup>.

In conclusion, we have to admit that the state of the art of formal language theory does not entirely satisfy the need of a powerful and practical formal grammar model, capable of accurately defining the entire range of constructs found in technical languages. Context-free grammars are the best available compromise between expressivity and simplicity. The compiler designer will supplement their weaknesses by other methods and tools, termed semantic, coming from general-purpose programming methods. They will be introduced in Chapter 6.

---

<sup>30</sup> An extreme case is a type 1 grammar presented in [47] to generate prime numbers encoded in unary base, i.e., the language  $\{a^n \mid n \text{ prime number}\}$ .

<sup>31</sup> For historical honesty, we mention that context-sensitive grammars have been occasionally considered by language and compiler designers. The language Algol 68 has been defined with a special class of type 1 grammars termed 2-level grammars, also known as VW-grammars from Van Wijngarten [55]; see also Cleaveland and Uzgalis [13].



<http://www.springer.com/978-1-84882-049-4>

Formal Languages and Compilation

Crespi Reghizzi, S.

2009, XII, 368 p. 100 illus., Hardcover

ISBN: 978-1-84882-049-4