

# 1 Languages

This book expresses and studies computation by using various languages, ranging from binary languages through programming languages to natural languages. The present chapter, consisting of three sections, formalizes languages so as to encompass all this range.

Section 1.1 defines languages and some fundamental operations. Section 1.2 introduces basic language generators, including expressions and grammars, and illustrates their use by defining a new programming language. Finally, Section 1.3 discusses translations of languages.

## 1.1 Formalization of Languages

This section formalizes languages and then introduces several language operations.

### Alphabets and words

To define languages, alphabets and words are first formalized.

#### Definition — alphabet and symbol

An *alphabet* is a finite, nonempty set of elements, which are called *symbols*. ◆

A sequence of symbols forms a word. The *empty word*, denoted by  $\varepsilon$ , is the word that contains no symbols. The next definition formally introduces words over an alphabet by using the recursive definitional method (see Section 0.2).

#### Definition — word

Let  $\Sigma$  be an alphabet.

1.  $\varepsilon$  is a word over  $\Sigma$ .
2. If  $x$  is a word over  $\Sigma$  and  $a \in \Sigma$ , then  $xa$  is a word over  $\Sigma$ . ◆

## Convention

In the theory of languages, *word* is synonymous with *string*. This book uses the former throughout. ●

The length of  $x$  is the number of all symbols in  $x$ .

## Definition — length of word

Let  $x$  be a word over an alphabet,  $\Sigma$ . The *length* of  $x$ ,  $|x|$ , is defined as follows:

1. if  $x = \varepsilon$ , then  $|x| = 0$
2. if  $x = a_1 \dots a_n$ , for some  $n \geq 1$ , where  $a_i \in \Sigma$  for all  $i = 1, \dots, n$ , then  $|x| = n$ . ◆

Let  $a \in \Sigma$ . Then,  $\#_a x$  denotes the number of occurrences of  $a$  in  $x$ .

## Operations on words

The following definitions introduce some basic operations over words.

## Definition — concatenation of words

Let  $x$  and  $y$  be two words over an alphabet,  $\Sigma$ . Then,  $xy$  is the *concatenation* of  $x$  and  $y$ . ◆

For every word  $x$ ,

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

## Definition — power of word

Let  $x$  be a word over an alphabet,  $\Sigma$ . For  $i \geq 0$ , the  *$i$ th power* of  $x$ ,  $x^i$ , is recursively defined as

1.  $x^0 = \varepsilon$
2.  $x^i = xx^{i-1}$ , for  $i \geq 1$ . ◆

Observe that for any word  $x$

$$x^i x^j = x^j x^i = x^{i+j}$$

where  $i, j \geq 0$ .

The reversal of a word is  $x$  written in the reverse order.

## Definition — reversal of word

Let  $x$  be a word over an alphabet,  $\Sigma$ . The *reversal* of  $x$ ,  $\text{reversal}(x)$ , is defined as

1. if  $x = \varepsilon$ , then  $\text{reversal}(x) = \varepsilon$
2. if  $x = a_1 \dots a_n$ , for some  $n \geq 1$ , and  $a_i \in \Sigma$ , for  $i = 1, \dots, n$ , then  $\text{reversal}(a_1 \dots a_n) = a_n \dots a_1$ .



## Definition — prefix of word

Let  $x$  and  $y$  be two words over an alphabet,  $\Sigma$ . Then,  $x$  is a *prefix* of  $y$  if there exists a word,  $z$ , over  $\Sigma$  so  $xz = y$ ; moreover, if  $x \notin \{\varepsilon, y\}$ , then  $x$  is a *proper prefix* of  $y$ .



For a word  $y$ ,  $\text{prefix}(y)$  denotes the set of all prefixes of  $y$ ; that is,

$$\text{prefix}(y) = \{ x : x \text{ is a prefix of } y \}$$

## Definition — suffix of word

Let  $x$  and  $y$  be two words over an alphabet,  $\Sigma$ . Then,  $x$  is a *suffix* of  $y$  if there exists a word,  $z$ , over  $\Sigma$  so  $zx = y$ ; moreover, if  $x \notin \{\varepsilon, y\}$ , then  $x$  is a *proper suffix* of  $y$ .



For a word  $y$ ,  $\text{suffix}(y)$  denotes the set of all suffixes of  $y$ ; that is,

$$\text{suffix}(y) = \{ x : x \text{ is a suffix of } y \}$$

## Definition — subword

Let  $x$  and  $y$  be two words over an alphabet,  $\Sigma$ . Then,  $x$  is a *subword* of  $y$  if there exist two word,  $z$  and  $z'$ , over  $\Sigma$  so  $zxz' = y$ ; moreover, if  $x \notin \{\varepsilon, y\}$ , then  $x$  is a *proper subword* of  $y$ .



For a word  $y$ ,  $\text{subword}(y)$  denotes the set of all subwords of  $y$ ; that is,

$$\text{subword}(y) = \{ x : x \text{ is a subword of } y \}$$

Observe that, for every word  $y$ , these three properties hold:

1.  $\text{prefix}(y) \subseteq \text{subword}(y)$
2.  $\text{suffix}(y) \subseteq \text{subword}(y)$
3.  $\{\varepsilon, y\} \subseteq \text{prefix}(y) \cap \text{suffix}(y) \cap \text{subword}(y)$ .

### Example 1.1.1 Operations over words

This example illustrates some of the notions that the present section has introduced so far. Consider the *binary alphabet* – that is,

$$\{0, 1\}$$

Notice that

$$\varepsilon$$

$$1$$

$$010$$

are words over  $\{0, 1\}$ . Observe that

$$|\varepsilon| = 0$$

$$|1| = 1$$

$$|010| = 3$$

Furthermore, note that

$$\#_0 \varepsilon = 0$$

$$\#_0 1 = 0$$

$$\#_0 010 = 2$$

The concatenation of 1 and 010 equals

$$1010$$

The fourth power of 1010 equals

$$10101010101010$$

Notice that

$$\text{reversal}(1010) = 0101$$

The words 10 and 1010 are prefixes of 1010. 10 is a proper prefix of 1010, whereas 1010 is not. Observe that

$$\text{prefix}(1010) = \{\varepsilon, 1, 10, 101, 1010\}$$

The words 010 and  $\varepsilon$  are suffixes of 1010. 010 is a proper prefix of 1010, whereas  $\varepsilon$  is not. Notice that

$$\text{suffix}(1010) = \{\varepsilon, 0, 10, 010, 1010\}$$

The words 01 and 1010 are subwords of 1010. 01 is a proper subword of 1010, but 1010 is not. Note that 01 is neither a prefix of 1010 nor a suffix of 1010. Finally, observe that

$$\text{subword}(1010) = \{\varepsilon, 0, 1, 01, 10, 010, 101, 1010\}$$



## Languages

Consider an alphabet,  $\Sigma$ . Let  $\Sigma^*$  denote the set of all words over  $\Sigma$ . Set

$$\Sigma^+ = \Sigma^* - \{\varepsilon\};$$

in other words,  $\Sigma^+$  denotes the set of all nonempty words over  $\Sigma$ . The following definition formalizes a language over  $\Sigma$  as a set of words over  $\Sigma$ . Notice that this definition encompasses both artificial languages, such as Pascal, and natural languages, such as English.

### Definition — language

Let  $\Sigma$  be an alphabet, and let  $L \subseteq \Sigma^*$ . Then,  $L$  is a *language* over  $\Sigma$ .



By this definition,  $\emptyset$  and  $\{\varepsilon\}$  are languages over any alphabet. Notice, however, that

$$\emptyset \neq \{\varepsilon\}$$

because  $\emptyset$  contains no element, while  $\{\varepsilon\}$  has one element, namely  $\varepsilon$ . Observe that for every alphabet,  $\Sigma$ ,  $\Sigma^*$  represents a language over  $\Sigma$ ; as  $\Sigma^*$  consists of all words over  $\Sigma$ , this language is referred to as the *universal language* over  $\Sigma$ .

Because languages are defined as sets, the notions concerning sets apply to them (see Section 0.1). Consequently, a language,  $L$ , is finite if it has  $n$  members, for some  $n \geq 0$ .

### Definition — finite and infinite language

Let  $L$  be a language.  $L$  is *finite* if  $\text{card}(L) = n$ , for some  $n \geq 0$ ; otherwise,  $L$  is *infinite*.



## Operations on languages

Consider the set operations of union, intersection, and difference (see Section 0.1). Naturally, these operations apply to languages. That is, for two languages,  $L_1$  and  $L_2$ ,

$$L_1 \cup L_2 = \{x: x \in L_1 \text{ or } x \in L_2\}$$

$$L_1 \cap L_2 = \{x: x \in L_1 \text{ and } x \in L_2\}$$

$$L_1 - L_2 = \{x: x \in L_1 \text{ and } x \notin L_2\}$$

Furthermore, consider a language  $L$  over an alphabet,  $\Sigma$ . The *complement* of  $L$ ,  $\bar{L}$ , is defined as

$$\bar{L} = \Sigma^* - L$$

The present section has already introduced several operations on words. The following definitions extend these word operations to languages.

### Definition — concatenation of languages

Let  $L_1$  and  $L_2$  be two languages. The *concatenation* of  $L_1$  and  $L_2$ ,  $L_1L_2$ , is defined as

$$L_1L_2 = \{xy: x \in L_1 \text{ and } y \in L_2\}$$



By this definition, every language  $L$  satisfies these two properties

1.  $L\{\varepsilon\} = \{\varepsilon\}L = L$
2.  $L\emptyset = \emptyset L = \emptyset$ .

### Definition — reversal of language

Let  $L$  be a language. The *reversal* of  $L$ ,  $\text{reversal}(L)$ , is defined as

$$\text{reversal}(L) = \{\text{reversal}(x): x \in L\}$$



### Definition — power of language

Let  $L$  be a language. For  $i \geq 0$ , the  *$i$ th power* of  $L$ ,  $L^i$ , is defined as

1.  $L^0 = \varepsilon$
2. for all  $i \geq 1$ ,  $L^i = LL^{i-1}$ .



## Definition — closure of language

Let  $L$  be a language. The *closure* of  $L$ ,  $L^*$ , is defined as

$$L^* = \bigcup_{i=0}^{\infty} L^i$$



## Definition — positive closure of language

Let  $L$  be a language. The *positive closure* of  $L$ ,  $L^+$ , is defined as

$$L^+ = \bigcup_{i=1}^{\infty} L^i$$



By the previous two definitions, for every language  $L$ , these two properties hold:

1.  $L^+ = LL^* = L^*L$
2.  $L^* = L^+ \cup \{\varepsilon\}$ .

The next example illustrates the set operations that this section has introduced so far. Besides the notations given in the previous definitions, this example also use these two notations

$$\text{prefix}(L) = \{ y: y \in \text{prefix}(x) \text{ for some } x \in L \}$$

$$\text{suffix}(L) = \{ y: y \in \text{suffix}(x) \text{ for some } x \in L \}$$

## Example 1.1.2 Operations over languages

Let  $\Sigma = \{0, 1\}$ . Consider these two languages over  $\Sigma$ :

$$L_1 = \{ 0, 01 \} \text{ and } L_2 = \{ 1, 01 \}$$

Observe that

$$L_1 \cup L_2 = \{ 0, 1, 01 \}$$

$$L_1 \cap L_2 = \{ 01 \}$$

$$L_1 - L_2 = \{ 0 \}$$

$$L_1 L_2 = \{ 01, 001, 011, 0101 \}$$

Furthermore, consider

$$L = \{ 10, 11 \}$$

over  $\Sigma$ . Notice that

$$\begin{aligned}\bar{L} &= \Sigma^* - \{10, 11\} \\ \text{reversal}(L) &= \{01, 11\} \\ \text{prefix}(L) &= \{\varepsilon, 1, 10, 11\} \\ \text{suffix}(L) &= \{\varepsilon, 0, 1, 10, 11\}\end{aligned}$$

For  $i = 2$ ,

$$L^2 = \{1010, 1011, 1110, 1111\}$$

Observe that

$$L^* = \{\varepsilon, 10, 11, 1010, 1011, 1110, 1111, \dots\}$$

and

$$L^+ = \{10, 11, 1010, 1011, 1110, 1111, \dots\}$$



## Models for languages and their investigation

Various kinds of computation can be described and studied by using languages. This book concentrates on the examination of models for languages because these models actually represent models of computation described by these languages. Naturally, the same language can be described by many different models; models of this kind are known as *equivalent models*.

To examine language models systematically, these models are classified according to their expressive power; then, the resulting *classes of models* are investigated. Most importantly, this investigation determines the *families of languages* characterized by these classes. If several different classes characterize the same language family, then these classes of models *have the same power*. Frequently, there is a need to demonstrate that some equally powerful classes contain no model that can perform a given computational task, specified by a language  $L$ . In terms of languages, this demonstration consists in proving that the language family characterized by these classes does not contain  $L$ . As a rule, such a proof is based on *pumping lemmas*. *Closure properties* are also discussed in detail. To explain closure properties, consider a language family  $L$  and a language operation  $\circ$ . If  $L$  contains every language resulting from the application of  $\circ$  to any languages in  $L$ , then  $L$  is *closed* under  $\circ$ ; otherwise,  $L$  is not closed under  $\circ$ .

### Example 1.1.3 Closure properties

Let  $L$  be the family of languages over  $\{a, b\}$  such that



$L \in \mathbf{L}$  if and only if each word in  $L$  begins with  $a$

That is,

$$\mathbf{L} = \{ L : L \subseteq \{a\}\{a, b\}^* \}$$

Notice that  $\mathbf{L}$  is closed under  $\cup$  because for any  $L, L' \in \mathbf{L}$ , each word in  $L \cup L'$  begins with  $a$ , so  $\mathbf{L}$  contains  $L \cup L'$ .

On the other hand,  $\mathbf{L}$  is not closed under reversal. Indeed, consider  $\{ab\} \in \mathbf{L}$  and observe that  $\text{reversal}(\{ab\}) = \{ba\}$ . As  $\{ba\} \notin \mathbf{L}$ ,  $\mathbf{L}$  is not closed under reversal. ▲

## 1.2 Expressions and Grammars

As pointed out in the conclusion of the previous section, the specification of languages represents an important topic. Finite languages can be specified by listing its components; for instance, the language consisting of all English articles is defined as

$$\{a, \text{an}, \text{the}\}$$

However, infinite languages cannot be described in this way. Therefore, special finite *metalanguages* – that is, languages that specify other languages – are used to generate infinite languages. This section introduces two language generators of this kind, expressions and grammars. First, Section 1.2.1 describes expressions, then Section 1.2.2 discusses grammar, finally Section 1.2.3 uses these language generators to design a new programming language, called COLA.

### 1.2.1 Expressions

A typical programming language contains logically cohesive lexical entities, such as identifiers or integers, over an alphabet,  $\Sigma$ . These entities, called *lexemes*, are customarily specified by *expressions*, recursively defined as follows:

1.  $\emptyset$  is a regular expression denoting the empty set.
2.  $\varepsilon$  is a regular expression denoting  $\{\varepsilon\}$ .
3.  $a$ , where  $a \in \Sigma$ , is a regular expression denoting  $\{a\}$ .
4. If  $r$  and  $s$  are regular expressions denoting the languages  $R$  and  $S$ , respectively, then
  - (a)  $(r \cdot s)$  is a regular expression denoting  $RS$
  - (b)  $(r + s)$  is a regular expression denoting  $R \cup S$
  - (c)  $(r^*)$  is a regular expression denoting  $R^*$ .

Whenever no ambiguity arises, parentheses are omitted in expressions. In this way, the next example describes Pascal identifiers.

### Example 1.2.1.1 Identifiers

Consider Pascal identifiers, defined as arbitrarily long alphanumeric words that begin with a letter. Equivalently and concisely, Pascal identifiers can be specified by using the expression

$$\langle \text{letter} \rangle \langle \text{letter or digit} \rangle^*$$

where

$$\begin{aligned} \langle \text{letter} \rangle &= A + \dots + Z \\ \langle \text{letter or digit} \rangle &= A + \dots + Z + 0 + \dots + 9 \end{aligned}$$



Expressions, which are precisely called *regular expressions*, are studied in Section 3.1.

## 1.2.2 Grammars

The syntax of programming languages is usually described by specification tools based on grammars. This section presents the following three grammatically based specification tools for programming languages:

1. Backus-Naur form
2. extended Backus-Naur form
3. syntax graphs.

### Backus-Naur form

The Backus-Naur form contains two kinds of symbols, terminals and nonterminals. Terminals denote lexemes, whereas nonterminals represent syntactic constructs, such as expressions. The heart of the Backus-Naur form is a finite set of productions. Each production has the form

$$A \rightarrow x_1 | \dots | x_n$$

In  $A \rightarrow x_1 | \dots | x_n$ ,  $A$  is a nonterminal. This nonterminal, the left-hand side of  $A \rightarrow x_1 | \dots | x_n$ , represents the syntactic construct that this production defines. The right-hand side is  $x_1 | \dots | x_n$ , where  $x_i$  is a word consisting of terminals and nonterminals. The words  $x_1$  through  $x_n$  represent  $n$  alternative definitions of  $A$ .

### Example 1.2.2.1 Part 1 Backus-Naur form

Consider the Backus-Naur form defined by its three productions:

1.  $\langle \text{expression} \rangle \rightarrow \langle \text{term} \rangle | \langle \text{term} \rangle + \langle \text{expression} \rangle | \langle \text{term} \rangle - \langle \text{expression} \rangle$
2.  $\langle \text{term} \rangle \rightarrow \langle \text{factor} \rangle | \langle \text{factor} \rangle * \langle \text{term} \rangle | \langle \text{factor} \rangle / \langle \text{term} \rangle$
3.  $\langle \text{factor} \rangle \rightarrow i | (\langle \text{expression} \rangle)$

This form has terminals  $+$ ,  $-$ ,  $*$ ,  $/$ ,  $($ ,  $)$ , and  $i$ , where  $i$  denotes an identifier or an integer. Furthermore, this form has three nonterminals, i.e.  $\langle \text{expression} \rangle$ ,  $\langle \text{term} \rangle$ , and  $\langle \text{factor} \rangle$ .

△

The Backus-Naur form uses its productions to derive syntactically well-formed constructs. This derivation begins from a special nonterminal, called the start symbol, and consists of several derivation steps. The Backus-Naur form makes a derivation step, symbolically denoted by  $\Rightarrow$ , according to a production,  $A \rightarrow x_1 | \dots | x_n$ , so that in the derived word, an occurrence of  $A$  is replaced with  $x_i$ , for some  $i = 1, \dots, n$ . The derivation ends when only terminals appear in the derived word.

### Example 1.2.2.1 Part 2 Derivations

Return to the Backus-Naur form defined in part 1 of this example. The present part describes how this form derives

$$i + i * i$$

The derivation starts from  $\langle \text{expression} \rangle$ , which represents the start symbol. In the brackets, every step of this derivation specifies the applied definition, selected from all alternative definitions appearing on the right-hand side of the used production.

|   |   |
|---|---|
| $\langle \text{expression} \rangle \Rightarrow \langle \text{term} \rangle + \langle \text{expression} \rangle$ | $[\langle \text{expression} \rangle \rightarrow \langle \text{term} \rangle + \langle \text{expression} \rangle]$ |
| $\Rightarrow \langle \text{factor} \rangle + \langle \text{expression} \rangle$                                 | $[\langle \text{term} \rangle \rightarrow \langle \text{factor} \rangle]$   |
| $\Rightarrow i + \langle \text{expression} \rangle$   | $[\langle \text{factor} \rangle \rightarrow i]$   |
| $\Rightarrow i + \langle \text{term} \rangle$   | $[\langle \text{expression} \rangle \rightarrow \langle \text{term} \rangle]$                                     |
| $\Rightarrow i + \langle \text{factor} \rangle * \langle \text{term} \rangle$                                   | $[\langle \text{term} \rangle \rightarrow \langle \text{factor} \rangle * \langle \text{term} \rangle]$           |
| $\Rightarrow i + i * \langle \text{term} \rangle$   | $[\langle \text{factor} \rangle \rightarrow i]$   |
| $\Rightarrow i + i * \langle \text{factor} \rangle$   | $[\langle \text{term} \rangle \rightarrow \langle \text{factor} \rangle]$   |
| $\Rightarrow i + i * i$   | $[\langle \text{factor} \rangle \rightarrow i]$   |

Observe that the Backus-Naur form discussed in here generates arithmetic expressions.

△

## Extended Backus-Naur form

The extended Backus-Naur form extends the Backus-Naur form's productions by adding the following three options.

1. The right-hand side of a production may contain some optional parts, which are delimited by brackets, [ and ].
2. The right-hand side of a production may contain braces, { and }, to indicate a syntactic part that can be repeated any number times, including zero times.
3. The right-hand side of a production may contain  $\mid$  and  $\mid$  to indicate several options, which are separated by  $\mid$ .

Note that some versions of the extended Backus-Naur form use parentheses ( and ), instead of  $\mid$  and  $\mid$ . However, because many programming languages contain parentheses as lexemes, the use of parentheses sometimes causes ambiguity; therefore, this book uses  $\mid$  and  $\mid$ .

### Example 1.2.2.1 Part 3 Extended Backus-Naur form

Recall that part 1 presented the following Backus-Naur form:

1.  $\langle \text{expression} \rangle \rightarrow \langle \text{term} \rangle \mid \langle \text{term} \rangle + \langle \text{expression} \rangle \mid \langle \text{term} \rangle - \langle \text{expression} \rangle$
2.  $\langle \text{term} \rangle \rightarrow \langle \text{factor} \rangle \mid \langle \text{factor} \rangle * \langle \text{term} \rangle \mid \langle \text{factor} \rangle / \langle \text{term} \rangle$
3.  $\langle \text{factor} \rangle \rightarrow i \mid \langle \langle \text{expression} \rangle \rangle$

In terms of the extended Backus-Naur form, production 1 can equivalently be redefined as

1.  $\langle \text{expression} \rangle \rightarrow \langle \text{term} \rangle \{ \mid + \mid - \} \langle \text{term} \rangle$

Analogously, shorten the other productions. The resulting extended Backus-Naur form becomes

1.  $\langle \text{expression} \rangle \rightarrow \langle \text{term} \rangle \{ \mid + \mid - \} \langle \text{term} \rangle$
2.  $\langle \text{term} \rangle \rightarrow \langle \text{factor} \rangle \{ \mid * \mid / \} \langle \text{term} \rangle$
3.  $\langle \text{factor} \rangle \rightarrow i \mid \langle \langle \text{expression} \rangle \rangle$

Notice that this form is more succinct than the original Backus-Naur form.

△

## Syntax graph

As already pointed out, a production of the extended Backus-Naur form defines the structure of the syntactic unit denoted by the nonterminal that forms the left-hand side of the production. Such a production is represented by a *syntax graph*, which

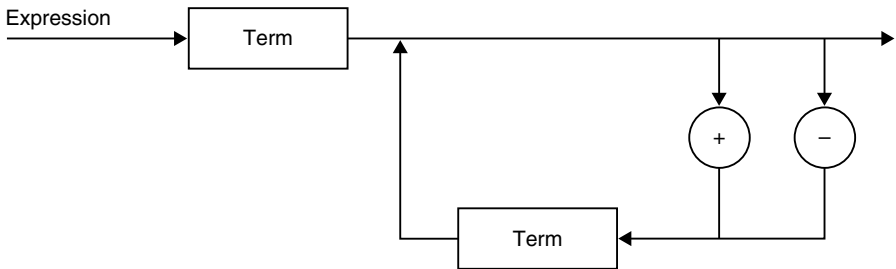
contains oval nodes and rectangular nodes. Oval nodes contain terminals, whereas rectangular nodes contain nonterminals. A syntax graph has an entering edge on the left and an exiting edge on the right. Each path that goes from the entering edge to the exiting edge gives rise to a valid structure of the syntactic unit defined by the graph.

### Example 1.2.2.1 Part 4 Syntax graph

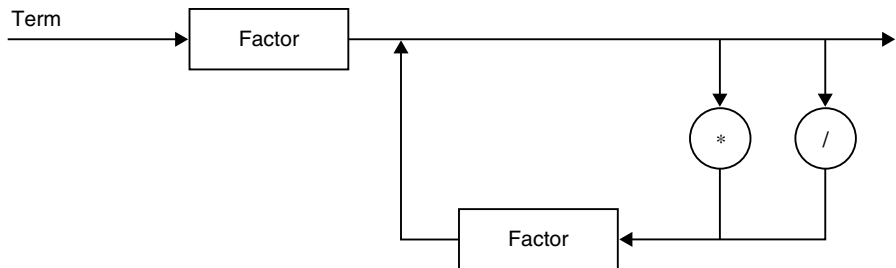
Part 3 gives the extended Backus-Naur form

1.  $\langle \text{expression} \rangle \rightarrow \langle \text{term} \rangle \{ [+|-] \langle \text{term} \rangle \}$
2.  $\langle \text{term} \rangle \rightarrow \langle \text{factor} \rangle \{ [*|/] \langle \text{term} \rangle \}$
3.  $\langle \text{factor} \rangle \rightarrow i | (\langle \text{expression} \rangle)$

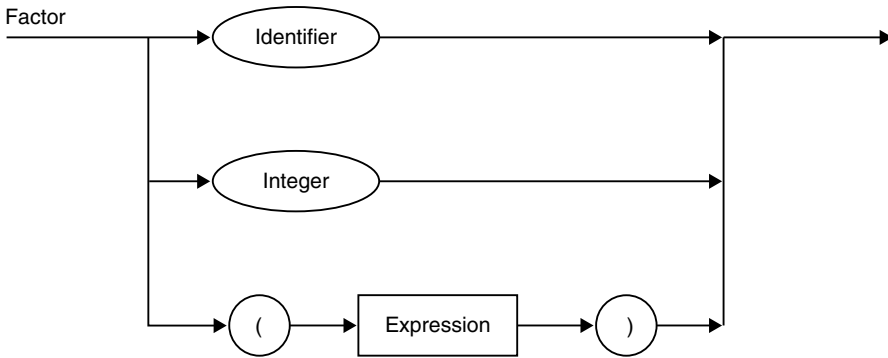
Figures 1.2.2.1 to 1.2.2.3 depict the syntax graphs corresponding to these three productions.



**Figure 1.2.2.1** Syntax graph corresponding to  $\langle \text{expression} \rangle \rightarrow \langle \text{term} \rangle \{ [+|-] \langle \text{term} \rangle \}$ .



**Figure 1.2.2.2** The syntax graph corresponding to  $\langle \text{term} \rangle \rightarrow \langle \text{factor} \rangle \{ [*|/] \langle \text{term} \rangle \}$ .



**Figure 1.2.2.3** The syntax graph corresponding to  $\langle \text{factor} \rangle \rightarrow i | ((\langle \text{expression} \rangle))$ .

△

## Polish notation

Parenthesized infix expressions, such as  $(3+1)*2$ , are often represented by *Polish notation*, which uses no parenthesis and, therefore, simplifies the evaluation of these expressions. Next, two fundamental kinds of this notation, the prefix Polish expressions and the postfix Polish expressions are recursively defined.

### Definition — prefix Polish expression

Let  $\Sigma$  be an alphabet, whose symbols denote operands. The *prefix Polish expressions* are defined recursively as follows:

1. If  $a$  is an infix expression,  $a \in \Sigma$ , then  $a$  is also the prefix Polish expression of  $a$ .
2. If  $U$  and  $V$  are infix expressions denoted by prefix Polish expressions  $X$  and  $Y$ , respectively, and  $o$  is an operator such that  $o \in \{+, -, *, /\}$ , then  $oXY$  is the prefix Polish expression denoting  $UoV$ .
3. If  $(U)$  is an infix expression, where  $U$  is denoted by the prefix Polish expression  $X$ , then  $X$  is the prefix Polish expression denoting  $(U)$ .

◆

### Definition — postfix Polish expression

Let  $\Sigma$  be an alphabet, whose symbols denote operands. The *postfix Polish expressions* are defined recursively as follows:

1. If  $a$  is an infix expression,  $a \in \Sigma$ , then  $a$  is also the postfix Polish expression of  $a$ .
2. If  $U$  and  $V$  are infix expressions denoted by prefix Polish expressions  $X$  and  $Y$ , respectively, and  $o$  is an operator such that  $o \in \{+, -, *, /\}$ , then  $XYo$  is the postfix Polish expression denoting  $UoV$ .
3. If  $(U)$  is an infix expression, where  $U$  is denoted by the postfix Polish expression  $X$ , then  $X$  is the postfix Polish expression denoting  $(U)$ .

◆

The evaluation of the prefix Polish expressions is left to the Exercises.

Next, is described the method of evaluating postfix Polish expressions. This consists of the following two steps, which are iterated until no operator appears in the given postfix expression.

1. Let  $E$  be the current postfix Polish expression; find the leftmost operator,  $o$ , appearing in  $E$  and the two operands,  $a$  and  $b$ , preceding  $o$ .
2. Perform the operation  $aob$  and replace  $abo$  in  $E$  with the obtained result.

### Example 1.2.2.1 Part 5 Postfix Polish notation

Consider the parenthesized infix expression

$$(3 + 1) * 2$$

Its postfix Polish equivalent is

$$31+2^*$$

The evaluation of  $31+2^*$  follows next.

*Iteration 1:*

1. The leftmost operator appearing in  $31+2^*$  is  $+$  and the two operands preceding this operator are 3 and 1.
2. Perform  $3 + 1$  to obtain 4, and replace  $31+$  with 4 in  $31+2^*$ ; the resulting expression has the form  $42^*$ .

*Iteration 2:*

1. The leftmost operator appearing in  $42^*$  is  $*$  and the two operands preceding this operator are 4 and 2.
2. Perform  $4 * 2$  to obtain 8, and replace  $42^*$  with 8.

Notice that this method correctly determines 8 as the resulting value of  $(3+1)*2$ .



This section has introduced three pragmatically oriented specification tools for programming language syntax, the Backus-Naur form, the extended Backus-Naur form, and syntax graphs. To investigate these tools in a rigorous way, Chapter 5 formalizes them using *context-free grammars*, which are systematically investigated in Part III.

## 1.2.3 Specification of a programming language

This section demonstrates the use of the language generators introduced in Sections

1.2.1 and 1.2.2 by designing a new *computer language* – COLA. This is a simple programming language, suitable for the evaluation of integer functions and sequences. To give an insight into COLA, consider the following COLA program:

```
begin
  read(n);
  write("resulting factorial", n, '! = ');
  factorial := 1;
  @iteration;
    if n = 0 goto @stop;
    factorial := factorial * n;
    n := n - 1;
  goto @iteration;
  @stop;
  write(factorial)
end
```

Although COLA has not been defined yet, it is intuitively clear that this program determines the factorial of *n*, where *n* is a nonnegative integer.

First, COLA lexemes are described by using expressions, discussed in Section 1.2.1. Then, COLA syntax is specified by using the extended Backus-Naur form and syntax graphs, introduced in Section 1.2.2.

## COLA lexemes

Next are specified the following five COLA lexemes by using expressions. Notice that each of these lexemes has an unbounded length.

1. identifiers
2. integers
3. labels
4. text literals
5. new-line text literals.

### *Identifiers*

COLA identifiers are nonempty alphanumeric words, which begin with a letter. Consequently, the COLA identifiers are specified by the expression

$$\langle \text{letter} \rangle \langle \text{letter or digit} \rangle^*$$

where  $\langle \text{letter} \rangle$  and  $\langle \text{letter or digit} \rangle$  are expressions defined as

$$\begin{aligned} \langle \text{letter} \rangle &= a + \dots + z \\ \langle \text{letter or digit} \rangle &= a + \dots + z + 0 + \dots + 9 \end{aligned}$$



*Integers*

COLA integers are nonempty numeric words. They are defined as

$$\langle \text{digit} \rangle \langle \text{digit} \rangle^*$$

where

$$\langle \text{digit} \rangle = 0 + \dots + 9$$

*Labels*

COLA labels have the form

$$@w$$

where  $w$  is a nonempty alphanumeric word; for instance, `@stop` is a well-formed COLA label. The COLA labels are defined by the expression

$$@ \langle \text{letter or digit} \rangle \langle \text{letter or digit} \rangle^*$$

where

$$\langle \text{letter or digit} \rangle = a + \dots + z + 0 + \dots + 9$$

*Text literals*

COLA text literals have the form

$$'w'$$

where  $w$  is a word consisting of any symbols except `'` or `"`; for instance, `'! ='` is a well-formed COLA text literal. The COLA text literals are defined by

$$' \langle \text{non-quotation symbol} \rangle^* '$$

where  $\langle \text{non-quotation symbol} \rangle$  denotes the set of all symbols except `'` or `"`.

*New-line text literals*

COLA new-line text literals have the form

$$"w"$$

where  $w$  is a word consisting of any symbols except `'` or `"`; for instance, `"resulting factorial"` is a validly formed COLA new-line text literal. COLA new-line text literals are defined by

$$" \langle \text{non-quotation symbol} \rangle^* "$$

where  $\langle \text{non-quotation symbol} \rangle$  denotes the set of all symbols except `'` or `"`.

As identifiers, integers, labels, text literals, and new-line text literals have an unbounded length, they are specified here by using expressions. Observe that all remaining COLA lexemes have a bounded length:

*arithmetic operators:* +, -, \*, /

*relational operators:* =, >, <

*parentheses:* (, )

*separators:* , and ;

*assignment operator:* :=

*reserved words:* **begin**, **end**, **goto**, **if**, **read**, **write**

## COLA syntax

COLA syntax is specified here by the extended Backus-Naur form and by syntax graphs.

The COLA lexemes represent terminals in the extended Backus-Naur form for COLA. Furthermore, this form has these nine nonterminals:

⟨expression⟩

⟨factor⟩

⟨program⟩

⟨read list⟩

⟨statement list⟩

⟨statement⟩

⟨term⟩

⟨write list⟩

⟨write member⟩

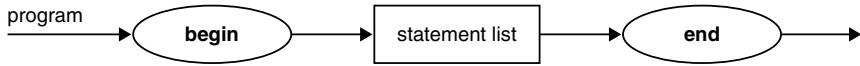
where ⟨program⟩ is the start symbol. The extended Backus-Naur form for COLA possesses the following nine productions:

1. ⟨program⟩ → **begin**⟨statement list⟩**end**
2. ⟨statement list⟩ → ⟨statement⟩{;⟨statement⟩}
3. ⟨statement⟩ → identifier := ⟨expression⟩  
                   **read**(⟨read list⟩)|  
                   **write**(⟨write list⟩)|  
                   [[if⟨expression⟩⟦>|<|=⟧⟨expression⟩] **goto**]label
4. ⟨read list⟩ → identifier{, identifier}
5. ⟨write list⟩ → ⟨write member⟩{,⟨write member⟩}
6. ⟨write member⟩ → ⟦ identifier|text literal|new-line text literal ⟧
7. ⟨expression⟩ → ⟨term⟩{ ⟦ +|- ⟧ <term⟩ }
8. ⟨term⟩ → ⟨factor⟩{ ⟦ \*|/ ⟧ ⟨factor⟩ }
9. ⟨factor⟩ → identifier |((⟨expression⟩))

Consider the first production

$$\langle \text{program} \rangle \rightarrow \text{begin} \langle \text{statement list} \rangle \text{end}$$

According to this production, a syntactically well-formed COLA program begins with **begin** and ends with **end**. Figure 1.2.3.1 presents the syntax graph visualizing this production.

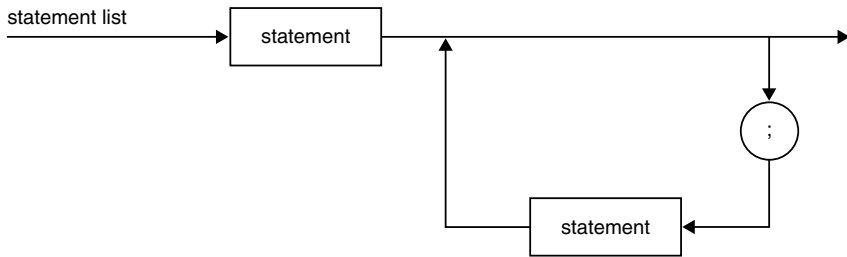


**Figure 1.2.3.1** Syntax graph corresponding to production 1.

The nonterminal  $\langle \text{statement list} \rangle$ , appearing on the right-hand side of production 1, forms the left-hand side of the second production:

$$\langle \text{statement list} \rangle \rightarrow \langle \text{statement} \rangle \{ ; \langle \text{statement} \rangle \}$$

This production indicates that a COLA statement list consists of a sequence of statements, separated by semicolons. Figure 1.2.3.2 gives the syntax graph displaying this production.



**Figure 1.2.3.2** Syntax graph corresponding to production 2.

Consider the third production,

$$\begin{aligned} \langle \text{statement} \rangle \rightarrow & \text{identifier} := \langle \text{expression} \rangle | \\ & \text{read}(\langle \text{read list} \rangle) | \\ & \text{write}(\langle \text{write list} \rangle) | \\ & [[\text{if} \langle \text{expression} \rangle \text{ '}' > | < | = } \langle \text{expression} \rangle ] \text{ goto } ] \text{label} \end{aligned}$$

The four alternative definitions appearing on the right-hand side of this production specify the following COLA statements.

1. The definition

$$\text{identifier} := \langle \text{expression} \rangle$$

specifies the COLA assignment statement; for instance,

$$\text{factorial} := \text{factorial} * n$$

is a valid COLA assignment statement. This instruction evaluates the expression appearing to the right of  $:=$  and assigns the resulting value to the identifier appearing to the left of  $:=$ .

2. The definition

**read**(⟨read list⟩)

specifies the COLA **read** statement; for instance,

**read**(n)

is a valid statement of this kind. This COLA **read** statement reads integers from the standard input and assigns these integer values to the members of the read list.

3. The definition

**write**(⟨write list⟩)

describes the COLA **write** statement; for instance,

**write**(factorial)

is a valid statement of this kind. This instruction writes the write list onto the standard output.

4. The definition

[[**if**(⟨expression⟩)  $\lceil > | < | = \rceil$  ⟨expression⟩] **goto**]label

contains three options:

- (a) label
  - (b) **goto** label
  - (c) **if**(⟨expression⟩)  $\lceil > | < | = \rceil$  ⟨expression⟩ **goto** label
- which are discussed below.

(a) The definition

label

specifies the COLA label statement; for instance,

@stop

is a valid statement of this kind. This instruction acts as a label used by the following two branch instructions.

(b) The definition

**goto** label

specifies the COLA unconditional branch statement; for instance,

**goto @stop**

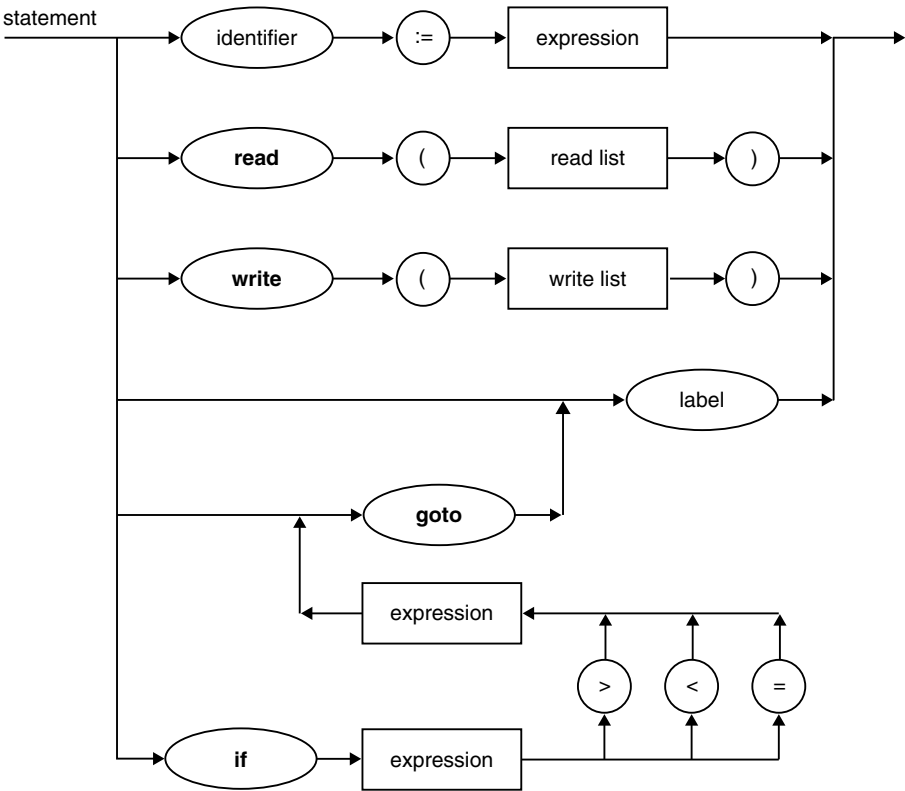
is a valid statement of this kind. This instruction causes the computation to continue at the label statement indicated by the label following **goto**.  
(c) The definition

**if**  $\langle \text{expression} \rangle \lceil > | < | = \rceil \langle \text{expression} \rangle$  **goto** label

specifies the COLA conditional branch statement; for instance,

**if**  $n = 0$  **goto** @stop

is a valid statement of this kind. This instruction compares the values of the two expressions by the relational operator appearing between these expressions. If this comparison holds true, the computation continues at the label statement indicated by the label following **goto**; otherwise, the instruction following this conditional branch statement is executed. Figure 1.2.3.3 presents the syntax graph displaying production 3.

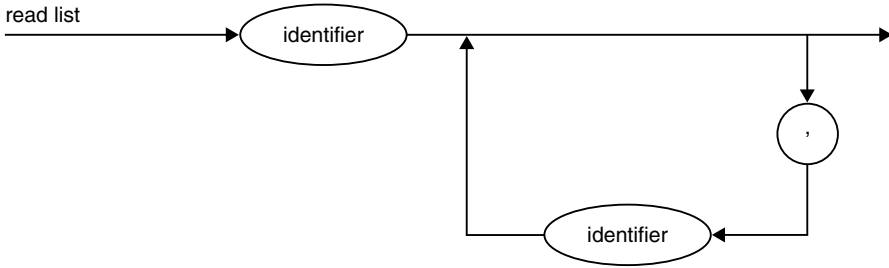


**Figure 1.2.3.3** Syntax graph corresponding to production 3.

Consider production 4:

$$\langle \text{read list} \rangle \rightarrow \text{identifier}\{, \text{identifier}\}$$

This production indicates that a COLA read list consists of a sequence of identifiers, separated by colons. Figure 1.2.3.4 presents the syntax graph depicting this production.

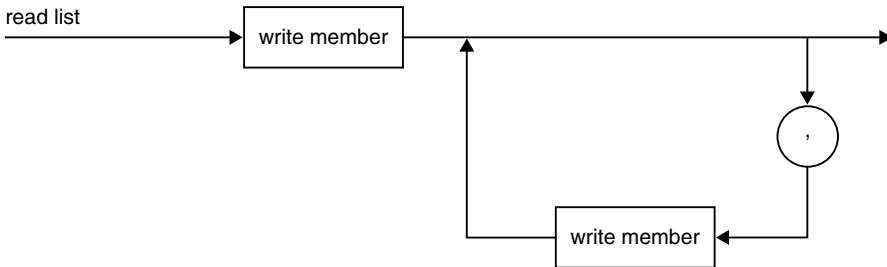


**Figure 1.2.3.4** Syntax graph corresponding to production 4.

The fifth production,

$$\langle \text{write list} \rangle \rightarrow \langle \text{write member} \rangle \{, \langle \text{write member} \rangle\}$$

implies that a COLA write list consists of a sequence of write member, separated by colons. Figure 1.2.3.5 presents the syntax graph depicting this production.



**Figure 1.2.3.5** Syntax graph corresponding to production 5.

The nonterminal  $\langle \text{write member} \rangle$ , appearing on the right-hand side of production 5, forms the left-hand side of the sixth production:

$$\langle \text{write member} \rangle \rightarrow [\text{identifier} | \text{text literal} | \text{new-line text literal}]$$

This production indicates that a write member is an identifier or a text literal or a new-line text literal. Figure 1.2.3.6 contains the corresponding syntax graph.

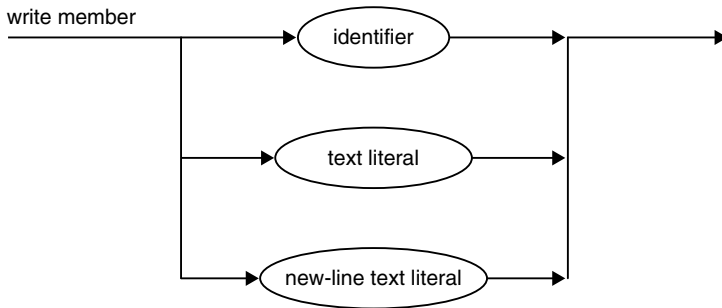
Finally, consider the remaining three productions 7–9:

$$\langle \text{expression} \rangle \rightarrow \langle \text{term} \rangle \{ \text{[ + | - ]} \langle \text{term} \rangle \}$$

$$\langle \text{term} \rangle \rightarrow \langle \text{factor} \rangle \{ \text{[ * | / ]} \langle \text{factor} \rangle \}$$

$$\langle \text{factor} \rangle \rightarrow \text{identifier} \mid (\langle \text{expression} \rangle)$$

Example 1.8, given in the previous section, has already discussed these productions and presented their syntax graphs (see Figures 1.2.2.1 to 1.2.2.3).



**Figure 1.2.3.6** Syntax graph corresponding to production 6.

The specification of COLA is complete. Reconsider the COLA program given in the beginning of this section; that is,

```

begin
  read(n);
  write("resulting factorial", n, '! =');
  factorial := 1;
  @iteration;
  if n = 0 goto @stop;
  factorial := factorial*n;
  n := n - 1;
  goto @iteration;
  @stop;
  write(factorial)
end
  
```

By the specification of COLA, this represents a well-formed COLA program as verified in the exercises.

## 1.3 Translations

Sections 1.1 and 1.2 formalized and discussed languages. This section studies translations between languages.

### Definition — translation

Let  $\Sigma$  be an *input alphabet*, and let  $\Omega$  be an *output alphabet*. A *translation*,  $\tau$ , from  $\Sigma^*$  to  $\Omega^*$  is a relation from  $\Sigma^*$  to  $\Omega^*$ .  $I_\tau$  is called the *input language* of  $\tau$  and denotes the domain of  $\tau$ ; that is,

$$I_\tau = \{ x: (x, y) \in \tau \text{ for some } y \in \Omega^* \}$$

$O_\tau$  is called the *output language* of  $\tau$  and denotes the range of  $\tau$ ; that is,

$$O_\tau = \{ y: (x, y) \in \tau \text{ for some } x \in \Sigma^* \}$$

For each  $x \in I_\tau$ ,  $\tau(x)$  is defined as

$$\tau(x) = \{ y: (x, y) \in \tau \}$$

For each  $L \subseteq I_\tau$ ,  $\tau(L)$  is defined as

$$\tau(L) = \{ y: (x, y) \in \tau \text{ for some } x \in L \}$$

Notice that for every translation  $\tau$  from  $\Sigma^*$  to  $\Omega^*$ ,

$$\tau(I_\tau) = O_\tau$$

Furthermore, for every language  $L$  such that  $L \subseteq I_\tau$ ,

$$\tau(L) \subseteq O_\tau$$

Observe that translations are defined as relations, so all notations concerning relations (reviewed in Section 0.2) apply to translations as well. For instance, a translation  $\tau$  is finite if  $\tau$  represents a finite relation.

### Example 1.3.1 Finite translation

This example describes a finite translation  $\tau$  that codes the decimal digits in binary. Formally, let  $\Sigma = \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10\}$ ,  $\Omega = \{0, 1\}$ , and  $\tau$  is the translation from  $\Sigma^*$  to  $\Omega^*$ , defined as

$$\tau = \{(0, 0), (1, 1), (2, 10), (3, 11), (4, 100), (5, 101), (6, 110), (7, 111), (8, 1000), (9, 1001)\}.$$

Observe that



$$I_\tau = \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}$$

$$O_\tau = \{0, 1, 10, 11, 100, 101, 110, 111, 1000, 1001\}$$

Notice that 8 is coded by  $\tau$  as

$$\tau(8) = \{1000\}$$

Furthermore, consider  $L = \{0, 2, 4, 6, 8\}$  and observe that

$$\tau(L) = \{0, 01, 100, 110, 1000\}$$



## Specification of translations

The specification of translations represents a similar problem to the specification of languages (see Section 1.2). As Example 1.3.1 illustrates, finite translations can be specified by listing their members. Naturally, this simple specification method is inapplicable to any infinite translations,  $\tau$ . If, however,  $\tau$  represents a special kind of infinite translation from  $\Sigma^*$  to  $\Omega^*$ , where  $\Sigma$  and  $\Omega$  are two alphabets, then the specification of  $\tau$  can be reduced to the specification of  $\tau(a)$ , for each  $a \in \Sigma$ . Substitution represents a translation of this kind.

### Definition — substitution

Let  $\Sigma$  and  $\Omega$  be two alphabets, and let  $\tau$  be a translation from  $\Sigma^*$  to  $\Omega^*$  such that for all  $x, y \in \Sigma^*$ ,

$$\tau(xy) = \tau(x)\tau(y).$$

Then,  $\tau$  is a *substitution*, and  $\tau^{-1}$  is an *inverse substitution*.



Consider a substitution  $\tau$  from  $\Sigma^*$  to  $\Omega^*$ . By the previous definition,  $\tau(\varepsilon) = \varepsilon$ . This definition also implies that  $\tau$  is completely specified by defining  $\tau(a)$ , for each  $a \in \Sigma$ . Indeed, for all  $x, y \in \Sigma^*$ ,  $\tau(xy) = \tau(x)\tau(y)$ , so for all nonempty words  $w$

$$\tau(w) = \tau(a_1) \dots \tau(a_n)$$

where  $|w| = n$ ,  $w = a_1 \dots a_n$ , and  $a_i \in \Sigma$  for  $i = 1, \dots, n$ .

### Example 1.3.2 Substitution

Consider the substitution  $\tau$  from  $\{0, 1\}^*$  to  $\{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}^*$  defined by

$$\tau(0) = \{0, 2, 4, 6, 8\} \text{ and } \tau(1) = \{1, 3, 5, 7, 9\}$$

Less formally,  $\tau$  transforms 0 and 1 to an even digit and an odd digit, respectively. As a result,  $\tau$  can be seen as a denary cryptography of binary information. Indeed, all denary numbers contained in  $\tau(w)$ , where  $w \in \{0, 1\}^*$ , encode  $w$ . For example, as  $\{5087, 1443\} \subseteq \tau(1001)$ , 5087 and 1443 belong to the denary numbers cryptographing 1001. As  $\tau^{-1}(\tau(w)) = w$ ,  $\tau^{-1}$  actually acts as a decoder of  $\tau$ . For example,  $\tau^{-1}$  decodes both 5087 and 1443 to 1001 because  $\tau^{-1}(5087) = 1001$  and  $\tau^{-1}(1443) = 1001$ .



A special case of substitution is homomorphism.

### Definition — homomorphism

Let  $\Sigma$  and  $\Omega$  be two alphabets, and let  $\tau$  be a substitution from  $\Sigma^*$  to  $\Omega^*$ . If  $\tau$  represents a function from  $\Sigma^*$  to  $\Omega^*$ , then  $\tau$  is a *homomorphism*, and  $\tau^{-1}$  is an *inverse homomorphism*.



### Example 1.3.3 Homomorphism

Morse code,  $\tau$ , represents an example of a homomorphism, mapping words consisting of Roman letters into  $\{., -\}^*$ :

$\tau(A) = \cdot -$   
 $\tau(B) = - \cdots$   
 $\tau(C) = - \cdot -$   
 $\tau(D) = - \cdots$   
 $\tau(E) = \cdot$   
 $\tau(F) = \cdots - \cdot$   
 $\tau(G) = - - \cdots$   
 $\tau(H) = \cdots \cdot$   
 $\tau(I) = \cdots$   
 $\tau(J) = \cdot - - -$   
 $\tau(K) = - \cdot -$   
 $\tau(L) = \cdot - \cdots$   
 $\tau(M) = - -$   
 $\tau(N) = \cdot \cdot$   
 $\tau(O) = - - -$   
 $\tau(P) = \cdot - - \cdot$   
 $\tau(Q) = - - \cdot -$   
 $\tau(R) = \cdot \cdot -$   
 $\tau(S) = \cdots \cdot$   
 $\tau(T) = -$   
 $\tau(U) = \cdot \cdot -$   
 $\tau(V) = \cdots -$

$\tau(W) = \text{'--}$   
 $\tau(X) = \text{'--'}$   
 $\tau(Y) = \text{'---}$   
 $\tau(Z) = \text{'---'}$



However, some infinite translations cannot be specified as simply as homomorphisms or substitutions; at this point, the language theory usually uses *translation grammars* to define them. Translation grammars resemble the Backus-Naur form (see Section 1.2.2). By analogy with the Backus-Naur form, translation grammars contain two kinds of symbols, terminals and nonterminals. Productions of translation grammars have the form

$$A \rightarrow x|y$$

### Example 1.3.4 Part 1 Translation grammar

Consider the translation grammar having the following eight productions:

$\langle \text{expression} \rangle \rightarrow \langle \text{expression} \rangle + \langle \text{term} \rangle \mid \langle \text{expression} \rangle \times \langle \text{term} \rangle +$   
 $\langle \text{expression} \rangle \rightarrow \langle \text{expression} \rangle - \langle \text{term} \rangle \mid \langle \text{expression} \rangle \times \langle \text{term} \rangle -$   
 $\langle \text{expression} \rangle \rightarrow \langle \text{term} \rangle \mid \langle \text{term} \rangle$   
 $\langle \text{term} \rangle \rightarrow \langle \text{term} \rangle * \langle \text{factor} \rangle \mid \langle \text{term} \rangle \times \langle \text{factor} \rangle *$   
 $\langle \text{term} \rangle \rightarrow \langle \text{term} \rangle / \langle \text{factor} \rangle \mid \langle \text{term} \rangle \times \langle \text{factor} \rangle /$   
 $\langle \text{term} \rangle \rightarrow \langle \text{factor} \rangle \mid \langle \text{factor} \rangle$   
 $\langle \text{factor} \rangle \rightarrow (\langle \text{expression} \rangle) \mid \langle \text{expression} \rangle$   
 $\langle \text{factor} \rangle \rightarrow i \mid i$

where  $i$  denotes an identifier or an integer. This grammar contains terminals  $i, +, -, *, /, ($ , and  $)$ . Furthermore, it has the nonterminals  $\langle \text{expression} \rangle$ ,  $\langle \text{factor} \rangle$ , and  $\langle \text{term} \rangle$ , where  $\langle \text{expression} \rangle$  is the start symbol.



Starting from a pair consisting of two start symbols, a translation grammar uses its productions to derive pairs of words over terminals; each step in this derivation is symbolically denoted by  $\Rightarrow$ . The set of all pairs derived in this way represents the translation defined by the grammar.

### Example 1.3.4 Part 2 Translation

The translation grammar given in the first part of this example translates infix arithmetic expressions to the equivalent postfix Polish expressions. Next, the present

example describes the derivation steps that translate  $i+i*i$  to  $iii^*+$ .

$\langle \text{expression} \rangle | \langle \text{expression} \rangle$   
 $\Rightarrow \langle \text{expression} \rangle + \langle \text{term} \rangle | \langle \text{expression} \rangle \langle \text{term} \rangle +$   
 $\Rightarrow \langle \text{term} \rangle + \langle \text{term} \rangle | \langle \text{term} \rangle \langle \text{term} \rangle +$   
 $\Rightarrow \langle \text{factor} \rangle + \langle \text{term} \rangle | \langle \text{factor} \rangle \langle \text{term} \rangle +$   
 $\Rightarrow i + \langle \text{term} \rangle | i \langle \text{term} \rangle +$   
 $\Rightarrow i + \langle \text{term} \rangle^* \langle \text{factor} \rangle | i \langle \text{term} \rangle \langle \text{factor} \rangle^* +$   
 $\Rightarrow i + \langle \text{factor} \rangle^* \langle \text{factor} \rangle | i \langle \text{factor} \rangle \langle \text{factor} \rangle^* +$   
 $\Rightarrow i + i^* \langle \text{factor} \rangle | ii \langle \text{factor} \rangle^* +$   
 $\Rightarrow i + i^* i | iii^* +$



Section 9.2.1 discusses translation grammars in greater detail. In addition, Section 9.3 demonstrates their use in practice. Specifically, based on these grammars, Section 9.3 constructs a complete compiler for the programming language COLA, described in Section 1.2.3.

## Exercises

*Note:* Making use of many formal notions introduced later on, Chapters 3 through 10 reconsider some of the following exercises in greater detail.

### 1 Formalization of Languages

#### 1.1 Consider the definition

Let  $\Sigma$  be an alphabet:

(a)  $\varepsilon$  is a word over  $\Sigma$

(b) if  $x$  is a word over  $\Sigma$  and  $a \in \Sigma$ , then  $ax$  is a word over  $\Sigma$ .

Section 1.1 has defined the words over  $\Sigma$  in a slightly different way. Are both definitions equivalent?

#### 1.2 Prove that for all words $x$

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

#### 1.3 Prove or disprove that for all $i \geq 0$ ,

$$\varepsilon^i = \varepsilon$$

#### 1.4 Prove or disprove that there exists a nonnegative integer $i$ such that for all words $x$

$$x^i = \varepsilon$$

- 1.5 Prove or disprove that for any two words,  $x$  and  $y$ ,

$$xy = yx$$

- 1.6 Prove that concatenation is associative.

- 1.7 Give a nonempty word  $x$  such that

$$x = \text{reversal}(x)$$

- 1.8 Give a nonempty word  $x$  such that

$$x^i = \text{reversal}(x)^i$$

for all  $i \geq 0$ .

- 1.9 Give a word  $x$  such that

$$x^i = \text{reversal}(x)$$

for all  $i \geq 0$ .

- 1.10 Prove that for every word  $x$

$$x^i x^j = x^j x^i = x^{i+j}$$

where  $i, j \geq 0$ .

- 1.11 Prove or disprove that for all words,  $x$  and  $y$ ,

$$\text{reversal}(xy) = \text{reversal}(y)\text{reversal}(x)$$

- 1.12 Let  $x = aaabababbb$ . Determine  $\text{prefix}(x)$ ,  $\text{suffix}(y)$ , and  $\text{subword}(x)$ .

- 1.13 Prove that every word  $y$  satisfies these three properties

(a)  $\text{prefix}(y) \subseteq \text{subword}(y)$

(b)  $\text{suffix}(y) \subseteq \text{subword}(y)$

(c)  $\{\varepsilon, y\} \subseteq \text{prefix}(y) \cap \text{suffix}(y) \cap \text{subword}(y)$

- 1.14 Let  $\Sigma = \{0, 1\}$ . Consider the language

$$L = \{011, 111, 110\}$$

over  $\Sigma$ . Determine  $\bar{L}$ ,  $\text{reversal}(L)$ ,  $\text{prefix}(L)$ ,  $\text{suffix}(L)$ ,  $L^2$ ,  $L^*$ , and  $L^+$ .

- 1.15 Prove that the following four properties hold for every language,  $L$ :

(a)  $L\{\varepsilon\} = \{\varepsilon\}L = L$

(b)  $L\emptyset = \emptyset L = \emptyset$

(c)  $L^+ = LL^* = L^*L$

(d)  $L^* = L^+ \cup \{\varepsilon\}$ .

- 1.16 Let  $L$  be a language. Characterize when  $L^* = L$ .

- 1.17 Define an enumerable language by analogy with the definition of an enumerable set.
- 1.18 Let  $\Sigma$  be an alphabet. Consider the family of all finite languages over  $\Sigma$ . Prove that this family is enumerable.
- 1.19\* Let  $\Sigma$  be an alphabet. Prove that  $2^{\Sigma^*}$  is not enumerable.
- 1.20 State the fundamental reason why some languages cannot be defined by any finite-size specification tools. (Section 8.1.5 discusses this crucial statement and its consequences in greater depth.)
- 1.21 Let  $\Sigma = \{0, 1\}$ . Consider the following two languages over  $\Sigma$ :

$$L_1 = \{00, 11\} \text{ and } L_2 = \{0, 00\}.$$

Determine  $L_1 \cup L_2$ ,  $L_1 \cap L_2$ ,  $L_1 - L_2$ , and  $L_1 L_2$ .

- 1.22 Prove that the following two equations hold for any two languages,  $L_1$  and  $L_2$ :
- (a)  $(L_1 \cup L_2)^* = (L_1^* L_2^*)^*$
- (b)  $(L_1 L_2 \cup L_2)^* = L_1 (L_2 L_1 \cup L_1)^*$
- 1.23 Prove that the following three equations hold for any three languages  $L_1$ ,  $L_2$ , and  $L_3$ :
- (a)  $(L_1 L_2) L_3 = L_1 (L_2 L_3)$
- (b)  $(L_1 \cup L_2) L_3 = L_1 L_3 \cup L_2 L_3$
- (c)  $L_3 (L_1 \cup L_2) = L_3 L_1 \cup L_3 L_2$
- 1.24 Prove or disprove that the following three equations hold for any three languages  $L_1$ ,  $L_2$ , and  $L_3$ :
- (a)  $(L_1 \cap L_2) L_3 = L_1 L_3 \cap L_2 L_3$
- (b)  $L_3 (L_1 \cap L_2) = L_3 L_1 \cap L_3 L_2$
- (c)  $L_3 (L_1 - L_2) = L_3 L_1 - L_3 L_2$
- 1.25 Determine all languages  $L_1$  and  $L_2$  satisfying

$$L_1 \cup L_2 = L_1 \cap L_2 = L_1 - L_2$$

- 1.26\* By *DeMorgan's law*,

$$\overline{L_1 \cup L_2} = \overline{L_1} \cap \overline{L_2}$$

for any two languages,  $L_1$  and  $L_2$ . Prove this law.

- 1.27 Example 1.1.3 discussed this family of languages

$$\mathbf{L} = \{ L : L \subseteq \{a\}^* \{a, b\}^* \}$$

Consider each language operation defined in Section 1.1. Prove or disprove that  $\mathbf{L}$  is closed under the operation.

- 1.28 Consider this family of languages

$$\mathbf{L} = \{ L : L \subseteq \{a, b\}^* \{b\}^* \{a, b\}^* \}$$

Consider each language operation defined in Section 1.1. Prove or disprove that  $L$  is closed under the operation.

- 1.29 Consider the family of languages

$$L = \{ L : L \subseteq \{a, b\}^*, \text{ and } |x| \text{ is even for each } x \in L \}$$

Consider each language operation defined in Section 1.1. Prove or disprove that  $L$  is closed under the operation.

- 1.30 Consider the family of finite languages. In addition, consider each language operation defined in Section 1.1. Prove or disprove that this family is closed under the operation.
- 1.31\* Consider the family of enumerable languages. In addition, consider each language operation defined in Section 1.1. Prove or disprove that this family is closed under the operation.
- 1.32\* Consider the family of infinite languages, which properly contains the family of enumerable languages. In addition, consider each language operation defined in Section 1.1. Prove or disprove that this family is closed under the operation.

## 2. Expressions and Grammars

- 2.1 Consider the finite language consisting of all English determiners, such as *any* and *some*. Specify this language by listing all its members.
- 2.2 Consider the expression

$$\langle \text{letter} \rangle \langle \text{letter or digit} \rangle^* \langle \text{letter} \rangle$$

where  $\langle \text{letter} \rangle = A + \dots + Z$ , and  $\langle \text{letter or digit} \rangle = A + \dots + Z + 0 + \dots + 9$ . Give an informal description of lexemes defined by this expression.

- 2.3 Construct expressions for all Pascal lexemes.
- 2.4 Let  $L$  be a language defined by an expression. Prove or disprove that any subset of  $L$  can be defined by an expression, too.
- 2.5 Select an infinite subset of English and define it by Backus-Naur form.
- 2.6 Construct the Backus-Naur form for the language consisting of the numbers in FORTRAN. By using this form, derive each of the following numbers:
- (a) 16
  - (b) -61
  - (c) -6.12
  - (d) -32.61E+04
  - (e) -21.32E-02.
- 2.7 Construct the Backus-Naur form for the language consisting of all PL/I declaration statements. By using this form, derive each of the following statements:
- (a) DECLARE A FIXED BINARY, B FLOAT

- (b) DECLARE (A, B) FIXED  
 (c) DECLARE (A(10), B(-1:2), C) FLOAT.
- 2.8 Construct the Backus-Naur form for the language of parenthesized logical expressions consisting of the logical variable  $p$  and the logical operators **and**, **or**, and **not**. By using this form, derive each of the following statements:
- (a) **not**  $p$  **or**  $p$   
 (b) **not** ( $p$  **and**  $p$ ) **or**  $p$   
 (c) ( $p$  **or**  $p$ ) **or** ( $p$  **and** **not**  $p$ ).
- 2.9 Construct the extended Backus-Naur form that specifies the Pascal syntax.
- 2.10 Construct syntax graphs that specify the Pascal syntax.
- 2.11 Prove that the following three specification tools define the same family of language.
- (a) the Backus-Naur form  
 (b) the extended Backus-Naur form  
 (c) syntax graphs
- 2.12 Modify the Backus-Naur form so
- (a) the right-hand side of any production contains only one definition  
 (b) a nonterminal may form the left-hand side of several productions.
- Formalize this modification and demonstrate its equivalence to the original Backus-Naur form.
- 2.13 Example 1.2.2.1 Part 1 discussed this three-production Backus-Naur form:

$$\begin{aligned} \langle \text{expression} \rangle &\rightarrow \langle \text{term} \rangle | \langle \text{term} \rangle + \langle \text{expression} \rangle | \langle \text{term} \rangle - \langle \text{expression} \rangle \\ \langle \text{term} \rangle &\rightarrow \langle \text{factor} \rangle | \langle \text{factor} \rangle * \langle \text{term} \rangle | \langle \text{factor} \rangle / \langle \text{term} \rangle \\ \langle \text{factor} \rangle &\rightarrow i | (\langle \text{expression} \rangle) \end{aligned}$$

Apply the modification discussed in Exercise 2.12 to this form.

- 2.14 Incorporate the Pascal relational and logical operators in the Backus-Naur form discussed in Part 1 of Example 1.2.2.1.
- 2.15 Make the extended Backus-Naur form more succinct by introducing some new notational options.
- 2.16 As noted in Section 1.2.2, some versions of the extended Backus-Naur form use ( **and** ) instead of  $\lceil$  and  $\rfloor$ , respectively. Demonstrate a practical disadvantage of the use of ( **and** ).
- 2.17 Section 1.2.2 has described syntax graphs informally. Define them rigorously.
- 2.18 Consider the infix expression  $(1+4)*((4+2)*3)+6$ . Determine and evaluate its postfix Polish equivalent.
- 2.19 Describe a method that evaluates the prefix Polish expressions.
- 2.20 Consider the infix expression  $7*((3+1)*8)+2$ . Determine and evaluate its prefix Polish equivalent.
- 2.21 Consider the COLA program given in the beginning of Section 1.2.3. By the specification of COLA given in Section 1.2.3, demonstrate that this program represents a well-formed COLA program.



### 3. Translations

- 3.1 Specify the finite translation  $\tau$  that translates the digits 0 through 9 to their octal representations.
- 3.2 Let  $\Sigma$  and  $\Omega$  be two alphabets, and let  $\tau$  be a substitution from  $\Sigma^*$  to  $\Omega^*$ . Does there exist a word,  $x \in \Omega^*$ , such that  $\tau^{-1}(x)$  is infinite? Does there exist a word,  $x \in \Omega^*$ , such that  $\tau^{-1}(x) = \emptyset$ ? Does there exist a word,  $x \in \Omega^*$ , such that  $\tau^{-1}(x) = \{\varepsilon\}$ ?
- 3.3\* Let  $\Sigma$  and  $\Omega$  be two alphabets, and let  $\tau$  and  $\tau'$  be two homomorphisms from  $\Sigma^*$  to  $\Omega^*$ . If for all  $x \in \Sigma^*$ ,

$$\tau(x) = \tau'(x)$$

then  $\tau$  and  $\tau'$  are equal. Design a method that decides whether two homomorphisms are equal.

- 3.4 Section 1.3 has stated that every homomorphism is a substitution? Explain this statement in detail.
- 3.5 Section 1.3 has stated that for every substitution  $\tau$ ,  $\tau(\varepsilon) = \varepsilon$ . Explain this statement in detail.
- 3.6 Section 1.3 has stated that for all  $x, y \in \Sigma^*$ ,

$$\tau(xy) = \tau(x)\tau(y),$$

so, for all nonempty words  $w$

$$\tau(w) = \tau(a_1 \dots a_n) = \tau(a_1) \dots \tau(a_n).$$

where  $n \geq 1$ ,  $w = a_1 \dots a_n$ , and  $a_i \in \Sigma$  for  $i = 1, \dots, n$ . Explain this statement in detail.

- 3.7 Design a translation grammar that translates infix expressions to the equivalent prefix Polish expressions.
- 3.8 Recall that syntax graphs graphically represent the Backus-Naur form. Design an analogical graphical representation for translation grammars – *translation graphs*.

## Programming projects

### 1 Formalization of Languages

- 1.1 Consider each of the unary language operations introduced in Section 1.1. Write a program that reads a finite language  $L$ , applies this operation to  $L$ , and produces the languages resulting from this application.
- 1.2 Consider each of the binary language operations introduced in Section 1.1. Write a program that reads two finite language  $L$  and  $L'$ , applies this operation to  $L$

and  $L'$ , and produces the language resulting from this application.

- 1.3 Introduce a finite language  $L$  representing a dictionary. Write a program that provides insertion and deletion of words in  $L$ .

## 2 Expressions and Grammars

- 2.1 Design a data structure for representing expressions.
- 2.2 Design a data structure that represents the Backus-Naur form.
- 2.2 Consider the Backus-Naur form given in Part 1 of Example 1.2.2.1. Write a program that reads a natural number  $n$  and then generates all  $m$ -step derivations in this form, for  $m = 1, \dots, n$ .
- 2.3 Consider the Backus-Naur form given in Part 1 of Example 1.2.2.1. Write a program that reads a word  $x$  and decides whether  $x$  is an arithmetic expression generated by this form.
- 2.4 Write a program that evaluates prefix Polish expressions.
- 2.5 Write a program that evaluates postfix Polish expressions.

## 3 Translations

- 3.1 Design a data structure for representing a substitution.
- 3.2 Design a data structure for representing a translation grammar.
- 3.3 Consider the translation grammar given in Part 1 of Example 1.3.3. Write a program that reads a natural number  $n$  and then generates all  $m$ -step derivations in this form, for  $m = 1, \dots, n$ .
- 3.4 Write a program that reads a word  $x$  and decides whether  $x$  is a valid infix expression. If  $x$  is valid, the program translates  $x$  into its postfix equivalent by using the translation grammar given in Example 1.3.3.
- 3.5 Write a program that reads a word  $x$  and decides whether  $x$  is a valid infix expression. If  $x$  is valid, the program translates  $x$  into its prefix equivalent.