Computational Nonlinear Morphology

With Emphasis on Semitic Languages

George Anton Kiraz
Beth Mardutho: The Syriac Institute

CAMBRIDGE
UNIVERSITY PRESS

PUBLISHED BY THE PRESS SYNDICATE OF THE UNIVERSITY OF CAMBRIDGE
The Pitt Building, Trumpington Street, Cambridge, United Kingdom

CAMBRIDGE UNIVERSITY PRESS
The Edinburgh Building, Cambridge CB2 2RU, UK
40 West 20th Street, New York, NY 10011-4211, USA
477 Williamstown Road, Port Melbourne, VIC 3207, Australia
Ruiz de Alarcón 13, 28014 Madrid, Spain
Dock House, The Waterfront, Cape Town 8001, South Africa

http://www.cambridge.org

First published 2001

Printed in the United Kingdom at the University Press, Cambridge

Typeface Times Roman 10/12 pt.   *System* LaTeX 2_ε [TB]

A catalog record for this book is available from the British Library.

Library of Congress Cataloging in Publication Data
Kiraz, George Anton.
Computational nonlinear morphology : with emphasis on Semitic languages/George Anton Kiraz.
p. cm. – (Studies in natural language processing)
Includes bibliographical references and index.
ISBN 0-521-63196-3
1. Grammar, Comparative and general – Morphology – Data processing. 2. Semitic languages – Morphology – Data processing. I. Title. II. Series.
P241 .K538 2001
492′.045′0285 – dc21
2001022304

ISBN 0 521 63196 3 hardback

Contents

1 Introduction

aṣ-ṣarf *of words* the deriving of words one from another, *of winds* shifting from one direction to another, *of wine* drinking it.

al-Fayrûz Abâdî (1329–1414)
al-qâmûs al-muḥîṭ

aṣ-ṣarf The shifting a thing from one state, or condition, to another.

Lane's *Arabic-English Lexicon*

Morphology The science of form.

Oxford English Dictionary

mor·phol·o·gy A study and description of word formation in a language including inflection, derivation, and compounding.

Webster's Third

This book might have a wide audience: computational linguists, theoretical and applied linguists, Semitists, and – who knows – maybe Biblical scholars with interest in Semitic. This is a mixed blessing. While it may serve as an interdisciplinary text, it makes introducing the matter at hand an arduous task. Nevertheless, this chapter attempts to introduce linguistic preliminaries to the nonlinguist, some computational prerequisites to the noncomputer specialist, and the basics of Semitic morphology to the nonsemitist. (To amuse the disappointed reader, I resorted to using quotations at the beginning of each chapter and elsewhere, mostly from the classical Semitic grammatical tradition. I hope this does not prove to be a further disappointment!)

In the definition of terms below, use was made of Trask (1993) and Crystal (1994). It must be noted that what follows is not intended to be an exhaustive coverage of the topics at hand. It must also be stressed that linguists may not necessarily, and often would not, agree with many of the definitions given here (the day is still to come when linguists agree on a definition for what the term "word" denotes). Definitions are given here in the context of the current work.

1.1 Linguistic Preliminaries

It has long been claimed that the morphology of many languages lies within the expressiveness of a class of formal languages known as "regular languages," and computational morphologists have taken up this claim. This section is an introduction to morphology (Section 1.1.1) and regular languages (Section 1.1.2). Another class of formal languages, the class of context-free languages on which some morphotactic models rely, is introduced as well (Section 1.1.3).

1.1.1 Morphology

1.1.1.1 Basic Definitions

Morphology is the branch of grammar that deals with the internal structure of words (Matthews, 1974). Although linguists may argue for other definitions of morphology, they mostly agree that morphology is the study of meaningful parts of words (McCarthy, 1991). In the English word /boys/, for example, there are two meaningful units: {boy} and the plural marker {s}. Such units, called *morphemes*, are the smallest units of morphological analysis. (Morphemes are shown in braces, { }; and the phonological word in solidi, / /.)

Sometimes, morphemes are not easily detected. Like /boys/, the English word /men/ is also a plural noun, but the plural morpheme in this case is embedded in the vowel [e], as opposed to [a] in singular /man/. In fact, morphemes are considered to be abstract units such as {PLURAL}. The {PLURAL} morpheme is realized in various forms called *morphs*: [s] in /boys/ and the vowel [e] in /men/.

Morphs in turn are made of *segments*. For example, {boy} consists of the segments: [b], [o], and [y]. Unless it constitutes a morph, a segment is meaningless. (Segments are shown in brackets, [ ].)

The morpheme that gives the main meaning of the word, for example, {boy} in /boys/, is called the *stem* or *root*. A *free morpheme* can stand on its own. In such a case, the morpheme and the word will be one and the same, for example, the word /boy/ and the morpheme {boy}. A *bound morpheme* requires additional morphemes to form a word, for example, the plural morpheme {s}.

Morphemes that precede the stem or root are called *prefixes*, such as {un} in English /unusual/. Those that follow are called *suffixes*, such as {s} in /boys/. In some languages, a morpheme may consist of two portions, neither of which is meaningful on its own. The first portion acts as a prefix and the second as a suffix. Such morphemes are called *circumfixes*. For example, in the Syriac word /neqṭlūn/ "to kill – IMPF PL 3RD MASC," the circumfix is {ne-ūn} "PL 3RD MASC."

The inventory of all morphs in a language constitutes the morphological lexicon. A lexicon of English need not have entries for /move/, /moved/, /moving/, /cook/, /cooked/, /cooking/, and so on. It only needs to list the unique morphs {move}, {cook}, {ed}, and {ing}. The suffixes apply to {move}, {cook}, and other verbs as well.

The sequence of lexical entries that make up a word is the *lexical form* of the word. For example, the lexical form of /moved/ is {move}β{ed}, where β denotes a *boundary symbol* that separates lexical entries. The word itself as one sees it on paper (or as one hears it), for example, /moved/, is called the *surface form*.

One important issue in morphology is conditional changes in morphemes. As noted above, the English word /moved/ contains two morphemes: {move} and {ed}. However, the [e] in {move} is deleted once the two morphemes are joined together. In this case, the change is merely orthographic. In other cases, the change might be phonologically motivated. For example, the nasal [n] in the negative morpheme prefix {in} becomes [m] when followed by a labial such as [p]. Hence, English /inactive/ from {in}β{active}, but /impractical/ from {in}β{practical}. Such changes are expressed by *rewrite rules*, also called *productions*. The [n] to [m] change in the above case may be expressed by the rule

$$\text{n} \rightarrow \text{m} \,/\, \underline{\quad\;}\ \text{p}$$

which reads: [n] rewrites as [m] before [p].

How does one know that */edkill/, as oppose to /killed/, from the morphemes {kill} and {ed}, is invalid? The licit combinations of morphemes are expressed by another form of rewrite rules, which we shall call here *morphotactic rules* such as

$$\text{word} \rightarrow \text{stem suffix}$$

which reads: “word” rewrites as “root” followed by “suffix.” Rewrite rules will be introduced further in Section 1.1.2.3.

1.1.1.2 Linear versus Nonlinear Morphology

Apart from Syriac /neqṭlūn/, the examples given above share one characteristic. The lexical form of a particular word is a sequence of morphemes from the lexicon. For example, the analysis of English /unsuccessful/ produces the lexical form {un}β{success}β{ful}. Because the surface form is generated by the concatenation of the lexical morphemes in question, this type of morphology is called *concatenative* or *linear* morphology.

In many languages, linearity does not hold. Consider the Arabic verb /kutib/ “to write – PERF PASS.” This verb consists of at least two morphemes: the root {ktb} “notion of writing” and the vocalic sequence {ui} “PERF PASS.” The concatenation of the two morphemes, */ktbui/ or */uiktb/, does not produce the desired result. In this case, the morphemes are combined in a *nonconcatenative*, or *nonlinear*, manner. (It will be shown in the next chapter how a third somewhat abstract morpheme dictates the manner in which the root and vocalic sequence are joined.)

The most ubiquitous linguistic framework for describing nonlinear morphology is based on the autosegmental model as applied to phonology (Goldsmith, 1976). *Autosegmental phonology* offers a framework under which nonlinear phonological (and morphological) phenomena can be described. Tense in Ngbaka, a language

Table 1.1. *Ngbaka tense is marked by tone*

Verb	Tone
kpòlò	low
kpōlō	mid
kpòló	low-high
kpóló	high

of Zaire (the modern Republic of Congo), for example, is indicated by tone, which is considered a morpheme in its own right. Consider the data in Table 1.1 (Nida, 1949). Each verb consists of two autonomous morphemes: {kpolo} "to return" and the respective tense morpheme, which is indicated by a specific tone.

Under the autosegmental model, autonomous morphemes are graphically represented on separate *tiers* as shown in Fig. 1.1. Each morpheme sits on its own autonomous tier: The morpheme {kpolo} sits on the lower tier, while the various tone morphemes, {L} "low," {M} "mid," {LH} "low-high," and {H} "high," sit on the upper tier. *Association lines* link segments from one tier to another. A pair of tiers, linked by some association line, is called a *chart*.[1]

Association lines follow specific rules of association according to two stipulations. The first stipulation is the *Well-Formedness Condition*: All vowels are associated with at least one tone segment and all tone segments are associated with at least one vowel segment, and association lines must not cross. The autosegmental representations in Fig. 1.1 meet this condition. However, the ill-formed representations in Fig. 1.2 violate the Well-Formedness Condition: In Fig. 1.2(a), the last vowel segment is not associated with a tone segment. In Fig. 1.2(b), the first tone segment is not associated with a vowel. In Fig. 1.2(c), association lines cross.

The second stipulation is the language-specific *Association Convention*, which states: Only the rightmost member of a tier can be associated with more than one member of another tier. The association of one member of a tier to more than one member of another tier is called *spreading*, for example, the spreading of the tone morphemes {L}, {M}, and {H} in Fig. 1.1.

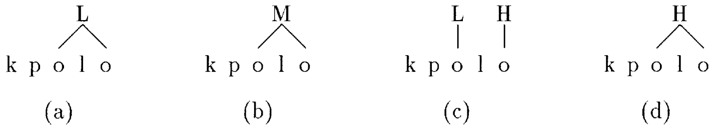

Fig. 1.1. Autosegmental representation of the Ngbaka tense in graphical form: (a) /kpòlò/, (b) /kpōlō/, (c) /kpòló/, (d) /kpóló/. Each morpheme sits on its own autonomous tier, with the stem on the lower tier and the respective tense tone morpheme on the upper tier.

[1] The term "chart" is mostly used in the computational linguistics literature, but not in the linguistic literature.

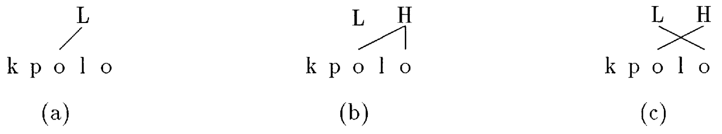

Fig. 1.2. Ill-formed autosegmental representations: (a) the last [o] segment is not linked; (b) the [L] tone segment is not linked; (c) association lines cross.

1.1.1.3 Between Phonology and Syntax

It was mentioned above that morphology is the branch of grammar that deals with the internal structure of words. Two other branches of grammar interact with morphology: phonology and syntax. The former concerns itself with the study of the sound system of languages, while the latter deals with the rules under which words combine to make sentences. Hence, phonology deals with units smaller than morphemes, while syntax describes units larger than words.

One rarely speaks of morphology without reference to phonology. (The term *morphophonology* denotes the phonological structure of morphemes.) One important aspect of phonology, which can hardly be separated from any morphological analysis of words, is *phonological processes*. These are conditional changes that alter segments. Some of the processes mentioned in this book are as follows: *assimilation*, in which one segment becomes identical to, or more like, another as in [n] → [m] above (see p. 3); *syncope*, or *deletion*, as the deletion of the first [a] in Syriac */qaṭal/ → /qṭal/[2] "to kill"; *epenthesis*, or *insertion*, as the insertion of / ʔi/ in Arabic /nkatab/ → / ʔinkatab/[3] "to write – REFL"; and *gemination*, or *doubling*, which involves the repetition of a segment (usually consonant) as in Arabic /katab/ → /kattab/[4] "to write – CAUS"; in this case, the gemination of [t] is morphologically motivated.

Another phonological phenomenon that concerns us is syllabification. The English word /morphology/, for example, consists of the syllables (separated by dots): mor·pho·lo·gy. *Open* syllables end in a vowel, for example, /lo/, while *closed* syllables end in a consonant, for example, /mor/. The components of a syllable can be represented by a smaller unit, the mora, for example, /lo/ consists of one mora while /mor/ consists of two morae; syllabic weight is defined by the number of morae in a syllable: *light* syllables contain one mora, while *heavy* syllables contain two morae.

One also rarely speaks of morphology without reference to syntax. It is not uncommon for an orthographic word in one language to represent a sentence in another. For example, Syriac /baytå/ "the house," /bbaytå/ "in the house," /dabbaytå/ "he who is in the house," /ldabbaytå/ "to him who is in the house," /waldabbaytå/ "and to him who is in the house" (Robinson, 1978). Syntax (apart from what is

[2] Syriac does not allow unstressed short vowels in open syllables, apart from few diachronic cases, for which see p. 115.
[3] Arabic is devoid of initial consonantal clusters.
[4] The Arabic causative is derived by the gemination of the second consonant.

required by morphotactics) is beyond the scope of this work. It suffices to note that for many languages, such as Semitic, the analysis of the orthographic word ventures into the realm of morphosyntax. In practical computational systems, a morphology module must account for phonology and – to some extent – syntax.

1.1.2 Regular Languages

Formal language theory establishes a hierarchy of formal languages based on their complexity and expressiveness. The class of regular languages is the most basic in the hierarchy.

Formal languages are defined in terms of *strings*, strings in terms of *alphabets*, and alphabets in terms of *sets*. These terms are introduced below.

1.1.2.1 Sets

A *set* is a collection of objects without repetition. A set can be specified by listing its objects. The following set represents the days of the week:

{ Monday, Tuesday, Wednesday, Thursday, Saturday, Sunday, Friday }

Each object in the set is called an *element* of the set. Elements are separated by a comma and are placed in braces, { }. No two elements can be the same; however, the order of the elements is not important. For instance, in the above set, Friday appears after Sunday. When the elements in the set are too long to list, one can use a *defining property* instead. The above set can be rewritten as follows:

$\{ x \mid x \text{ is a weekday} \}$

Read: x where x is a weekday.

If an element x is a member of a set A, we say $x \in A$ (read: x in A). If an element x is not a member of a set A, we say $x \notin A$ (read: x not in A). For example, given the set $A = \{1, 2, 5\}$, we say $2 \in A$, but $3 \notin A$.

The set containing no elements, usually denoted by { } or $\emptyset$, is called the *empty set*.

A set A is a *subset* of another set B, designated by $A \subset B$, if every element in A is an element in B. For example, { 1,2 } is a subset of { 1,2,3,4 }; however, { 1,5 } is not a subset of { 1,2,3,4 } because the latter does not include the element 5. If A is a subset of B but may also be equal to B, we say $A \subseteq B$.

There are several operations that can be applied to sets: The *union* of sets A and B, denoted by $A \cup B$, is the set that consists of all the elements in either A or B. For example, let $A = \{1, 2, 3\}$ and $B = \{3, 4, 5\}$, then $A \cup B = \{1, 2, 3, 4, 5\}$. Note that since a set cannot have duplicates, the union contains only one instance of the element 3. We write

$$\bigcup_{i=1}^{n} A_i$$

to denote $A_1 \cup A_2 \cup \cdots \cup A_n$.

The *intersection* of sets A and B, denoted by $A \cap B$, is the set that consists of all the common elements in A and B. For example, let $A = \{1, 2, 3\}$ and $B = \{3, 4, 5\}$; then $A \cap B = \{3\}$. We write

$$\bigcap_{i=1}^{n} A_i$$

to denote $A_1 \cap A_2 \cap \cdots \cap A_n$.

The *difference* of sets A and B, denoted by $A - B$, is the set that consists of all the elements in A that are not in B. For example, let $A = \{1, 2, 3\}$ and $B = \{3, 4, 5\}$; then $A - B = \{1, 2\}$.

The *complement* of a set A, denoted by $\overline{A}$, is the set that consists of all the elements in the universe that are not in A. The *universe* set contains all elements under consideration. If we assume that the universe set contains all the days of the week and

$$A = \{\text{Monday, Wednesday, Friday}\}$$

then

$$\overline{A} = \{\text{Tuesday, Thursday, Saturday, Sunday}\}$$

The *cross product* of sets A and B, denoted by $A \times B$, is a set consisting of all the pairs (a_1, a_2) where the first element, a_1, is in A_1 and the second element, a_2, is in A_2. For example, let $A = \{1, 2\}$ and $B = \{3, 4, 5\}$; then $A \times B = \{(1, 3), (1, 4), (1, 5), (2, 3), (2, 4), (2, 5)\}$. We write

$$\prod_{i=1}^{n} A_i$$

to denote $A_1 \times A_2 \times \cdots \times A_n$. We also write B^n to denote the cross product of B by itself n times; that is, $\underbrace{B \times B \times \cdots \times B}_{n \text{ times}}$.

With the use of defining properties, the above operations can be defined as follows:

$$\begin{aligned}
A \cup B &= \{x \mid x \in A \text{ or } x \in B\} \\
A \cap B &= \{x \mid x \in A \text{ and } x \in B\} \\
A - B &= \{x \mid x \in A \text{ but } x \notin B\} \\
\overline{A} &= \{x \mid x \notin A\} \\
A \times B &= \{(a_1, a_2) \mid a_1 \in A_1, a_2 \in A_2\}
\end{aligned}$$

A *finite* set contains a finite number of elements. For instance, the set $\{n \mid 1 \le n \le 10\}$ is a finite set of 10 elements, that is, the integers 1 to 10. An *infinite* set contains an infinite number of elements. For example, $\{n \mid 1 \le n\}$ represents all positive integers, from 1 to infinity.

Any subset of the cross product $A_1 \times A_2$ is called a *binary relation*. A_1 is called the *domain* of the relation and A_2 is called the *range* of the relation. It is possible to have a relation on one set, for example, a relation on $A \times A$.

1.1.2.2 Alphabets and Strings

An *alphabet* is a finite set of symbols. *Symbols* are usually letters or characters. The English alphabet is the set

$$\{\,\text{A,B}, \ldots, \text{Y,Z,a,b}, \ldots, \text{y,z}\,\}$$

A *string over* some alphabet is a finite sequence of elements drawn from that alphabet. If $A = \{\text{a,b,c}\}$ is an alphabet, then the following sequences, *inter alia*, are strings over A: "a," "aa," "aab," "aac," "caa," and "cbbba." However, the string "aad" is not a string over A since the element 'd' is not in A. (Strings are shown in double quotes when they appear in text; characters or symbols are shown in single quotes.)

The number of elements in a string x determines the *length* of the string, denoted by $|x|$. The length of the string "ab" is two and the length of "cbba" is four. A string of length zero is called the *empty string* and is denoted by ϵ.

The terms prefix and suffix apply to strings as they apply to natural languages (see p. 2).

The *concatenation* of two strings x and y, denoted by xy, is the string formed by appending y to x. For example, if x is "may" and y is "be," then xy is "maybe."

Concatenation is used to define *exponentiation*. If x is a string, we write x^2 to denote the concatenation of x with itself twice, that is, xx. Similarly, x^3 denotes the concatenation of x with itself thrice, that is, xxx. In this vein, $x^1 = x$ and x^0 is the empty string ϵ. For example, let x be the string "ha;" we say x^0 is ϵ, x^1 is "ha," x^2 is "haha," x^3 is "hahaha," and so on.

The *Kleene star*, denoted by x^*, denotes zero or more concatenations of x with itself: ϵ, x, xx, xxx, and so on. To exclude the empty string, we use the Kleene plus notation, x^+, which denotes one or more concatenations of x with itself.

1.1.2.3 Languages, Expressions, and Grammars

The term *language*, or *formal language*, denotes any set of strings over some alphabet. For example, let $A = \{\,\text{a,b,c}\,\}$ be some alphabet. All of the following sets of strings are languages over A:

$$L_1 = \{\,\text{b,ab,aab,aaab,aaaab}, \ldots\}$$
$$L_2 = \{\,\text{b}\,\}$$
$$L_3 = \{\,\text{abcc,abca,aaba,ccca,caba}, \ldots\}$$

L_1 is an infinite language over A where each string consists of zero or more instances of ‘a’ followed by one instance of ‘b’. L_2 is a finite language over A, and it consists of only one string, the string being a symbol from the alphabet. L_3 is a finite language over A whose strings are of length four. The language { abc, add }, however, is not a language over A since “add” is not a string over A; this is so because ‘d’ is not in A.

Expressions are used to describe the strings of a language. The strings in L_1, for example, can be expressed by the expression a*b: zero or more instances of ‘a’ followed by one ‘b’. The language L_2 can be expressed by the expression b since it contains only that element. Expression may contain other operators such as disjunction, | (read ‘or’). For instance, the strings in L_3 begin with either an ‘a’ or a ‘c’, followed by three arbitrary symbols from A; this can be described by the expression $(\mathrm{a} \mid \mathrm{c})A^3$.

Given two alphabets, one can use expressions to describe languages over the two alphabets. Consider the following two alphabets, which represent capital and small letters, respectively:

$$C = \{\,\mathrm{A,B}, \ldots, \mathrm{Y,Z}\,\}$$
$$S = \{\,\mathrm{a,b}, \ldots, \mathrm{y,z}\,\}$$

The language CS^* consists of all strings that start with one capital letter followed by zero or more small letters, e.g. “I,” “Good,” “Bed.” The language

$$(C \mid S)S^3\,\mathrm{ing}$$

consists of all strings that start with either a capital or small letter, followed by three small letters, followed by “ing,” for example, “booking,” and “Writing.”

Languages are described by grammars. A *formal grammar* consists of an alphabet and a set of rewrite rules. Generally, a *rewrite rule* consists of a left-hand-side and a right-hand side separated by an arrow, for example,

$$\mathrm{y} \rightarrow \mathrm{i\ e}$$

Read: ‘y’ rewrites as ‘i’ followed by ‘e’. Applying the rule on the string “entrys,” which consists of the stem {entry} concatenated with the plural morpheme {s}, results in “entries,” after replacing ‘y’ by “ie.” This is the rule that applies to English plurals ending in a ‘y.’ However, there is nothing preventing the rule from applying to any ‘y.’ Applying the rule on “may” results in the undesired “maie.”

To restrict the application of rules, one specifies *left* and *right contexts*, separated by an environment bar ___ . The rule only applies when the contexts are satisfied. The above rule can be rewritten as

$$\mathrm{y} \rightarrow \mathrm{i\ e} \,/\, \_\_\_\ \mathrm{s}$$

Read: ‘y’ rewrites as ‘i’ followed by ‘e’ before ‘s.’ Here, the left context is not specified. (The slash, /, separates the contexts from the right-hand side.) The above

$$
\begin{array}{lcl}
\mathrm{S} & \longrightarrow & \text{the A} \\
\mathrm{A} & \longrightarrow & \text{old B} \\
\mathrm{B} & \longrightarrow & \text{man} \\
\mathrm{B} & \longrightarrow & \text{woman}
\end{array}
$$

Fig. 1.3. A set of rewrite rules that generate the sentences *the old man* and *the old woman*. Nonterminal symbols start with a capital letters.

rule only applies when there is an ‘s’ to the right of ‘y;’ hence, it does not apply to “may.”

So far, *terminal symbols* were used in rules; that is, symbols drawn from the alphabet in question. It is also possible to use *nonterminal symbols*; that is, symbols that are derived from other symbols. These are designated with capital letters. Consider the following alphabet whose symbols are actual words, {man, old, the, very, woman}, and the rules in Fig. 1.3. The first rule states that a sentence S rewrites as the word “the” followed by A. According to the second rule, the symbol A in turn, rewrites as the word “old” followed by B. Now B rewrites as either “man” or “woman” according to the third and fourth rules, respectively. This grammar generates the two sentences: “the old man” and “the old woman”. The derivations can be illustrated graphically by *parse trees* as in Fig. 1.4.

Grammars, and hence languages derived from them, are of various complexities. The least complex are *regular language*s. These can be generated by rewrite rules of the form

$$\mathrm{A} \rightarrow a\,\mathrm{B}$$

or

$$\mathrm{A} \rightarrow a$$

Here A and B are nonterminal symbols and a is a terminal symbol. The formal definition of regular languages over an alphabet Σ is as follows:

(i) The empty set is a regular language.
(ii) For each a in Σ, $\{\,a\,\}$ is a regular language.

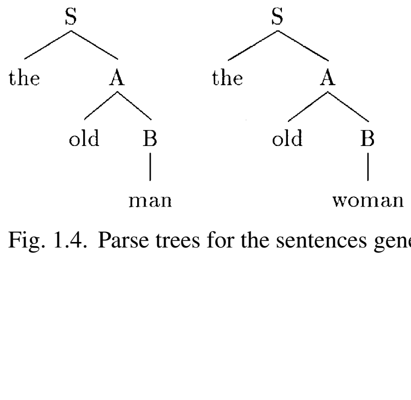

Fig. 1.4. Parse trees for the sentences generated by the rules in Fig. 1.3.

(iii) If L_1, L_2, and L are regular languages, then so are

$$\begin{array}{ll} L_1 L_2 = \{\, xy \mid x \in L_1,\, y \in L_2 \,\} & \text{(concatenation)} \\ L_1 \cup L_2 = \{\, x \mid x \in L_1 \text{ or } x \in L_2 \,\} & \text{(union)} \\ L^* = \bigcup_{i=0}^{\infty} L^i & \text{(Kleene star)} \end{array}$$

(iv) There are no other regular languages.

Every regular language is described by a regular expression. A *regular expression* over some alphabet is an expression consisting of a sequence of symbols from the alphabet constructed by means of concatenation, union, or Kleene star. For example, let { a,b,c } be an alphabet; then "ab" (concatenation), a ∪ b (union), and a* (Kleene star) are regular expressions. More formally, a regular expression over an alphabet Σ is defined as follows:

(i) ∅ is a regular expression and denotes the empty set.
(ii) ϵ is a regular expression and denotes the set $\{\, \epsilon \,\}$.
(iii) For each a in Σ, a is a regular expression and denotes the set $\{\, a \,\}$.
(iv) If r_1, r_2, and r are regular expressions denoting the languages R_1, R_2, and R, respectively, then so are
$r_1 r_2$, denoting $R_1 R_2$ (concatenation)
$r_1 \mid r_2$, denoting $R_1 \cup R_2$ (union)
r^*, denoting R^* (Kleene star)
(v) There are no other regular expressions.

Some languages cannot be expressed by regular grammars. Let 'a' and 'b' be two symbols in an alphabet. It is possible to express a language that contains a *specified* number of 'a's followed by the same number of 'b's, say $\mathrm{a}^3\mathrm{b}^3$, which denotes the language { aaabbb }. However, it is not possible to express a language that contains an *unspecified* number of 'a's followed by the same number of 'b's such as the expression $\mathrm{a}^n\mathrm{b}^n$, that is, n 'a's followed by n 'b's.

1.1.3 Context-Free Languages

A more expressive class of languages is the *context-free* family of languages, which can cope with $a^n b^n$ languages. Context-free languages are generated by context-free grammars.

1.1.3.1 Context-Free Grammars

Context-free languages are generated by rewrite rules called *context-free rules*. These may not contain left or right contexts as their name implies. However, unlike regular grammars, context-free rewrite rules may contain any number of nonterminals on the right-hand side of rules.

Consider the rules in Fig. 1.5, which generates the sentences "the book," "the books," "a book," and "some books." Figure 1.6 gives the parse trees of the generated sentences.

NP	$\longrightarrow$	Det SingNoun
NP	$\longrightarrow$	Det PlNoun
NP	$\longrightarrow$	SingDet SingNoun
NP	$\longrightarrow$	PlDet PlNoun
Det	$\longrightarrow$	the
SingDet	$\longrightarrow$	a
PlDet	$\longrightarrow$	some
SingNoun	$\longrightarrow$	book
PlNoun	$\longrightarrow$	books

Fig. 1.5. A context-free grammar. Nonterminals start with a capital letter.

The first four rules in Fig. 1.5 actually describe one linguistic phenomenon: a noun phrase (NP) rewrites as a determiner (Det) followed by a noun. The duplication in rules (with the prefixes 'Sing' and 'Pl' to indicate singular and plural, respectively) are necessary in order to ensure that the determiner and noun agree in number. Unification provides a mechanism by which such duplications can be avoided.

1.1.3.2 Unification

A *feature* represents a characteristic. For example, NUMBER is a feature of nouns. Similarly, PERSON is a feature of verbs. Typically, a feature has a *value*. The feature NUMBER, for example, takes the values sing and pl for singular and plural, respectively. (Features are shown in SMALL CAPS, while values are shown in small type.)

A matrix of features constitutes a *category*, for example,

$$\begin{bmatrix} \text{NUMBER} = \text{sing} \\ \text{PERSON} = \text{2nd} \\ \text{GENDER} = \text{masc} \end{bmatrix}$$

Unification is an operation that combines two categories into one as long as the two initial categories do not contain conflicting information. The resulting category

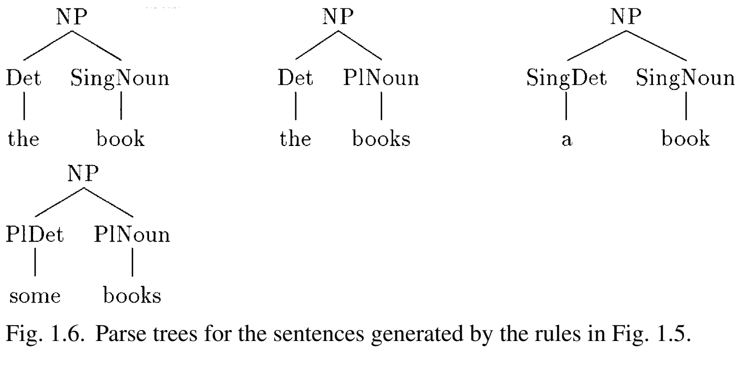

Fig. 1.6. Parse trees for the sentences generated by the rules in Fig. 1.5.

contains all the information in the two initial categories. As a way of illustration, let

$$A = \begin{bmatrix} \text{NUMBER} = \text{sing} \\ \text{PERSON} = \text{2nd} \end{bmatrix}, B = \begin{bmatrix} \text{PERSON} = \text{2nd} \\ \text{GENDER} = \text{masc} \end{bmatrix}$$

The unification of A and B produces a new category C,

$$C = \begin{bmatrix} \text{NUMBER} = \text{sing} \\ \text{PERSON} = \text{2nd} \\ \text{GENDER} = \text{masc} \end{bmatrix}$$

which combines all the information in A and B. Boxed indices, e.g. $\boxed{1}$, $\boxed{2}$, and so on, are used to avoid writing the same value again and again in different categories. For example, A and B above can be rewritten as

$$A = \begin{bmatrix} \text{NUMBER} = \text{sing} \\ \text{PERSON} = \text{2nd}\,\boxed{1} \end{bmatrix}, B = \begin{bmatrix} \text{PERSON} = \boxed{1} \\ \text{GENDER} = \text{masc} \end{bmatrix}$$

If the value of PERSON is unknown, but both A and B must have the same value, then the following can be written

$$A = \begin{bmatrix} \text{NUMBER} = \text{sing} \\ \text{PERSON} = \boxed{1} \end{bmatrix}, B = \begin{bmatrix} \text{PERSON} = \boxed{1} \\ \text{GENDER} = \text{masc} \end{bmatrix}$$

Now let

$$D = \begin{bmatrix} \text{NUMBER} = \text{sing} \\ \text{PERSON} = \text{2nd} \end{bmatrix}, E = \begin{bmatrix} \text{PERSON} = \text{1st} \\ \text{GENDER} = \text{masc} \end{bmatrix}$$

The unification of D and E fails since they contain contradicting information, that is, the value of the feature PERSON.

Unification is more powerful than the above examples show. The value of a feature may be a variable. Let

$$F = \begin{bmatrix} \text{NUMBER} = \text{sing} \\ \text{PERSON} = \text{X} \end{bmatrix}, G = \begin{bmatrix} \text{PERSON} = \text{1st} \\ \text{GENDER} = \text{Y} \end{bmatrix}$$

where X and Y are variables. The unification of F and G produces the following new category H,

$$H = \begin{bmatrix} \text{NUMBER} = \text{sing} \\ \text{PERSON} = \text{1st} \\ \text{GENDER} = \text{Y} \end{bmatrix}$$

where the value of PERSON was *instantiated* from G. The value of GENDER, however, remains uninstantiated.

When a feature can possibly take more than one value, set notation is used, for example,

$$I = \begin{bmatrix} \text{NUMBER} = \{\,\text{sing,plural}\,\} \\ \text{PERSON} = \text{1st} \\ \text{GENDER} = \text{Y} \end{bmatrix}$$

Unification allows the passing of information from one category to another. It is used in grammars, which gives rise to the term "unification-based" grammars.

1.1.3.3 Unification-Based Context-Free Grammars

The rules in Fig. 1.5 are repeated in Fig. 1.7, using unification. Here, all nonterminals are associated with a category indicating number. All categories present in a rule must unify. The first rule states that a noun phrase rewrites as a determiner followed by a noun, both of which must have the same value for NUMBER, that is, X. Hence, "a book" and "some books" are valid, while *"a books" is invalid. Since the determiner "the" can take singular and plural nouns, its NUMBER value is

$$\begin{array}{rcl}
\text{NP}\begin{bmatrix} \text{NUMBER} = \text{X} \end{bmatrix} & \rightarrow & \text{Det}\begin{bmatrix} \text{NUMBER} = \text{X} \end{bmatrix} \\
 & & \text{Noun}\begin{bmatrix} \text{NUMBER} = \text{X} \end{bmatrix} \\
\text{Det}\begin{bmatrix} \text{NUMBER} = \text{X} \end{bmatrix} & \rightarrow & \text{the} \\
\text{Det}\begin{bmatrix} \text{NUMBER} = \text{sing} \end{bmatrix} & \rightarrow & \text{a} \\
\text{Det}\begin{bmatrix} \text{NUMBER} = \text{pl} \end{bmatrix} & \rightarrow & \text{some} \\
\text{Noun}\begin{bmatrix} \text{NUMBER} = \text{sing} \end{bmatrix} & \rightarrow & \text{book} \\
\text{Noun}\begin{bmatrix} \text{NUMBER} = \text{pl} \end{bmatrix} & \rightarrow & \text{books}
\end{array}$$

Fig. 1.7. A repetition of the rules in Fig. 1.5, using unification-based context-free rules. Nonterminals are associated with a category indicating number.

a variable X. The value is instantiated from noun entries: sing in "the book" and pl in "the books."

1.2 Computational Preliminaries

It was mentioned that the claim that the morphology of many languages lies within the expressiveness of regular languages was adopted by computational morphologists. The main attraction of regular languages is the ease of modeling them by the simplest computational devices, known as "finite-state automata." This section introduces computational morphology and topics related to finite-state automata.

1.2.1 Computational Morphology

Computational morphology is a subfield of computational linguistics (also called "natural language processing" or "language engineering"). Computational morphology concerns itself with computer applications that analyze words in a given text, such as determining whether a given word is a verb or a noun. Consider, for example, a spelling checker. To find if a word is spelled correctly, the program searches a lexicon (a list of the words in that language) for the word in question. In order for the spelling checker to work, the lexicon must contain all the forms and inflections of each word (e.g., book, books, booked, booking, etc.), resulting in a huge lexicon with a few hundred thousand entries. This is even more dramatic in morphologically rich languages: some of the Arabic spelling checkers have a lexicon with over 10 million entries!

A more efficient and elegant spelling checker can be achieved by listing in the lexicon unique stems and morphemes, and having a morphological component of the program derive words. An Arabic spelling checker that makes heavy use of morphology should not contain more than 10,000 lexical entries[5] to cover Modern Standard Arabic, of course excluding personal names, foreign loans, and so on.

Almost all practical applications that deal with natural language must have a morphological component. After all, an application must first recognize the word in question before analyzing it syntactically, semantically, or whatever the case may be.

The typical morphological analyzer tackles three issues: the morphological lexicon, rewrite rules, and morphotactic rules. The lexicon encodes all the morphemes in a given language. Rewrite rules handle orthographic changes, phonological processes, and the like. Morphotactic rules determine which morphemes can be combined to form grammatical words.

[5] Wehr's dictionary of Modern Standard Arabic (Wehr, 1971) contains 6167 roots, 3014 of which are used in the derivation of both verb and noun stems (Daniel Ponsford, personal communication).

The state-of-the-art methodology in computational morphology for handling the lexicon and rewrite rules makes use of devices called finite-state automata. Morphotactic grammars can be described in two ways: either by using finite-state automata, especially for most purely concatenative languages, or unification-based context-free grammars for more complex languages. The rest of this section introduces finite-state automata and demonstrates their application in computational morphology.

1.2.2 Finite-State Automata

A finite-state automaton (FSA) is usually modeled by a program. The program receives a string from an input *tape*. It reads one character at a time from left to right. After reading the last character, it either accepts or rejects the string. An automaton that accepts English strings would accept the input /receive/ but would reject */recieve/. The terms "automaton" and "machine" will be used interchangeably.

A FSA consists of a finite number of *states*. Before scanning the first character from the input tape, the machine will be in a special *initial state*. At any point while scanning the input, the FSA will be in one particular state, called the *current state*. One or more states will be marked as *final states*. The FSA in Fig. 1.8(a),

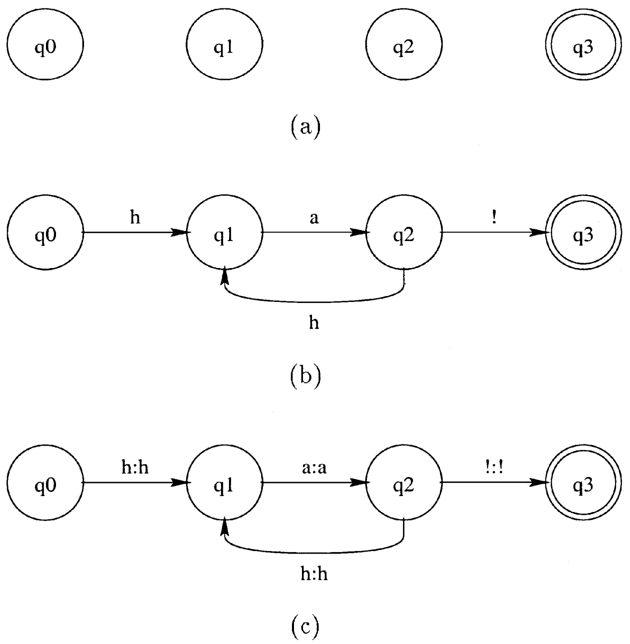

Fig. 1.8. A laughing automaton: (a) gives the states of the machine with final states marked by double circles; (b) adds transitions; (c) is the identity transducer of the automaton in (b) where each symbol is mapped to itself.

for example, consists of four states (represented by circles), labeled q_0, q_1, q_2, and q_3. By default we always assume that state q_0 is the initial state. Final states are indicated by double circles (e.g. q_3).

The program that represents the FSA consists of a set of instructions of the form (q_0, h, q_1), which is interpreted as follows: if the machine is currently in state q_0 and the next character to scan from the input tape is 'h,' then move to state q_1. Graphically, such an instruction is represented by a *transition*, an arrow labeled 'h' that goes from state q_0 to state q_1. The FSA in Fig. 1.8(a) is repeated in Fig. 1.8(b) with the following transitions: (q_0, h, q_1), (q_1, a, q_2), (q_2, h, q_1), (q_2, !, q_3).

A *deterministic* finite-state automaton does not have more than one transition leaving a state on the same label. An *ϵ-free automaton* does not contain any transitions labeled with the empty string ϵ.

After the last character of the input is scanned, if the automaton is in a final state, the input is *accepted*; otherwise, it is *rejected*. Consider the input string "ha!" and the automaton in Fig. 1.8(b). After reading the 'h,' the automaton moves from state q_0 to q_1. After reading the 'a,' it moves to state q_2. After reading the last character '!,' it moves to state q_3. Since there are no more characters in the input string and the machine is in a final state, the input "ha!" is accepted. In fact, the machine is a laughing machine, which accepts the strings "ha!," "haha!," "hahaha!," and so on (after Gazdar and Mellish, 1989). Now consider the input string "ha" (without the exclamation mark). After reading the first 'h' and the 'a,' the machine will be in state q_2. There are no more characters in the input string; however, state q_2 is *not* a final state. Hence, the string "ha" is rejected.

The set of strings that an automaton accepts is the *language* accepted by that machine. The language accepted by the above laughing machine is $L = \{$ha!, haha!, hahaha!, ...$\}$. Languages that can be described by an FSA belong to the class of regular languages, as opposed to the more powerful class of context-free languages (see Sections 1.1.2 and 1.1.3).

The result given by a FSA is limited: either the string is accepted, or it is rejected. Another form of FSAs is finite-state *transducers* (FSTs). An FST is a FSA, but instead of scanning one tape, it scans two tapes simultaneously. One string is usually designated as input and the other as output. Each transition is labeled with a pair: the first denotes the character on the first tape, and the second denotes the character on the second tape. Consider the FST in Fig. 1.9(a). It *transduces* (or maps) English laughter into French laughter. If we consider the first tape to be the input and the second the output, then the machine will transduce English 'ha!' into French 'ah!' as follows: after reading an 'h' on the first tape, it writes an 'a' on the second tape and enters state q_1. Similarly, reading 'a' from the first tape, it writes 'h' on the second tape ending in state q_2. Finally, the machines reads '!' from the first tape and writes the same character on the second tape. The transitions for transducing English "haha!" into French "ahah!" are shown schematically in Fig. 1.9(b). The numbers between the two tapes indicate the current state after scanning the input symbol in question.

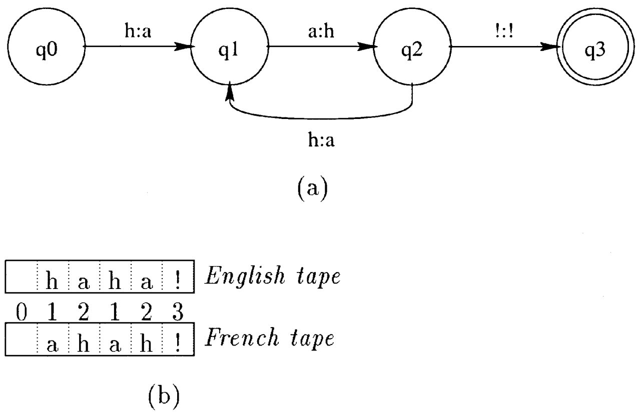

Fig. 1.9. An English–French laughing transducer: (a) gives the transition diagram that maps English laughter into French; (b) shows the transitions for the input "haha!," with the numbers between the two tapes indicating the current state after the input symbol in question is scanned.

The transducer that maps every symbol to itself is called the *identity transducer*. For example, the identity of the automaton in Fig. 1.8(b) is the transducer in Fig. 1.8(c). We denote the identity transducer of a machine A by $Id(A)$.

It was mentioned above (see p. 11) that every regular language is described by a regular expression. It was also mentioned in this section that languages that are described by FSAs are regular. The *Kleene correspondence theorem* shows the equivalence among regular languages, regular expressions, and finite-state automata as follows:

(i) Every regular expression describes a regular language.
(ii) Every regular language is described by a regular expression.
(iii) Every finite-state automaton accepts a regular language.
(iv) Every regular language is accepted by a finite-state automaton.

FSAa and FSTs are interesting for a few reasons. First, they are simple to model. The transitions in Fig. 1.8(b), for example, can be represented by a simple matrix as shown in Table 1.2.[6] Second, transducers are bidirectional. There is nothing stopping us from using the second tape as input and the first as output. In the case of the transducer in Fig. 1.9(a), this results in a French–English laughing transducer. Third, their closure properties (discussed next) allow the combination

[6] Representing an automaton with n states using a matrix, however, requires n^2 space, regardless of the number of transitions. A more efficient way would be to store transitions. For an example in C++, see Budd (1994, Section 16.5); for an example in Prolog, see Kiraz and Grimley-Evans (1997). Compression methods for large and sparse automata can be found in Aho, Sethi, and Ullman (1986, p. 144 ff.).

Table 1.2. *Transitions for Fig. 1.9(a)*

	q_0	q_1	q_2	q_3
q_0		h:a		
q_1			a:h	
q_2		h:a		!:!
q_3				

of various machines using operations such as concatenation, union, and so on to create more complex machines.

1.2.3 Regular Operations and Closure Properties

One performs operations on automata in the same manner by which one performs operations on sets, for example; union, intersection, and so on. Since FSAs represent regular languages and a language is merely a set of strings, one can intuitively deduce the result of such operations. The union of two automata A and B, for example, produces an automaton that accepts strings that are accepted by either A or B.

In addition to the operations mentioned in the formal definition of regular languages (i.e., concatenation, union, and Kleene star; see p. 10), one can define other operations that may be applied to FSAs or FSTs. Like the intersection of sets (see p. 7), the *intersection* of automata A and B, denoted by $A \cap B$, is the automaton that accepts strings that are accepted by both A and B. For example, if A describes the regular language $a \mid b \mid c^*$ and B describes the regular language $a^2 \mid b \mid c^+$, then $A \cap B$ accepts the language $b \mid c^+$.

The *difference* of automata A and B, denoted by $A - B$, is the automaton that accepts strings that are accepted by A, but not by B. For example, considering automata A and B from the previous example, then $A - B$ accepts the language $a \mid \epsilon$ (note that $c^* - c^+$ gives ϵ).

The *complement* of an automaton A, denoted by $\overline{A}$, is the automaton that accepts all strings (over some alphabet) apart from those in A. For example, let Σ be an alphabet and let A be an automaton; then $\overline{A} = \Sigma^* - A$.

Like the cross product of sets (see p. 7), the *cross product* of automata A and B, denoted by $A \times B$, is the transducer that maps the strings accepted by A into the strings that are accepted by B. For example, if A describes the regular language $ab \mid c$ and B describes the regular language $de \mid f$, then $A \times B$ produces the regular relation

$$\big((ab){:}(de)\big) \mid \big((ab){:}f\big) \mid \big(c{:}(de)\big) \mid \big(c{:}f\big)$$

One additional operation we shall encounter is *composition*, which is confined to FSTs. When two transducers are composed, the output of the first transducer is used as input to the second. For example, if T_1 is a transducer that maps a to b, and T_2 is another transducer that maps b to c, then their composition, denoted by $T_1 \circ T_2$, is the transducer that maps a to c.

A regular language is said to be *closed under* a specific operation if the application of the operation to any regular language results in another regular language. From the definition of regular languages (see p. 10), we say that regular languages, and their corresponding FSAs, are closed under concatenation, union, and Kleene star. This is so because the concatenation of any two regular languages – by definition – results in a regular language. The same holds for union and Kleene star.

FSAs and ϵ-free transducers are closed under intersection, difference, and complement. However, ϵ-containing transducers are not closed under these operations (Kaplan and Kay, 1994, p. 342). Transducers are closed under composition.

1.2.4 Finite-State Morphology

This section demonstrates how computational morphology makes use of regular languages, regular expressions, and finite-state automata. A typical morphological analyser handles lexica, rewrite rules, and morphotactic rules.

1.2.4.1 The Lexica

The morphological lexicon is the set of stems and morphemes in a language. Being a set of strings, it is usually represented with an automaton (Sproat, 1992, p. 128 ff.). Figure 1.10(a) gives an automaton for a small English lexicon representing the words /book/ (with transitions through states q_0–q_4), /hook/ (with transitions through the same states but with a different path), /move/ (states q_0, q_5–q_9), and /model/ (states q_0, q_5–q_{12}). Final states mark the end of a lexical entry. Note that entries that share prefixes (in the formal sense), such as "mo" in /move/ and /model/, share the same transitions for the prefix. Figure 1.10(b) gives another automaton for the suffixes {ed} and {ing}.

One way to combine the two machines in Fig. 1.10 is by concatenation, yielding a machine that accepts /booked/, /booking/, /hooked/, /hooking/, and so on. Usually however, one would want to separate the morphemes by a special boundary symbol, say β. A two-state machine that accepts this symbol is created and is concatenated between the two machines in Fig. 1.10, that is, $L_1 \beta L_2$. This machine accepts /bookβed/, /bookβing/, /hookβed/, /hookβing/, and the like.

1.2.4.2 Rewrite Rules

The finite-state approach is the most common method in computational morphology and phonology for modeling rewrite rules. Each rule is compiled, by some algorithm, into a finite-state transducer that performs the mapping desired by the rule. As a way of illustration, consider the derivation of /moving/ from the lexical