CHAPTER 3

Symbolic Natural Language Processing

3.0. Introduction

Fundamental notions of combinatorics on words underlie natural language processing. This is not surprising, since combinatorics on words can be seen as the formal study of sets of strings, and sets of strings are fundamental objects in language processing.

Indeed, language processing is obviously a matter of strings. A text or a discourse is a sequence[1] of sentences; a sentence is a sequence of words; a word is a sequence of letters. The most universal levels are those of sentence, word and letter (or phoneme), but intermediate levels exist, and can be crucial in

[1] In this chapter, we will not use the term "word" to denote a sequence of symbols, in order to avoid ambiguity with the linguistic meaning.

some languages, between word and letter: a level of morphological elements (e.g. suffixes), and the level of syllables. The discovery of this piling up of levels, and in particular of word level and phoneme level, delighted structuralist linguists in the 20th century. They termed this inherent, universal feature of human language as “double articulation”.

It is a little more intricate to see how *sets* of strings are involved. There are two main reasons. First, at a point in a linguistic flow of data being processed, you must be able to predict the set of possible continuations after what is already known, or at least to expect any continuation among some set of strings that depends on the language. Second, natural languages are ambiguous, i.e. a written or spoken portion of text can often be understood or analyzed in several ways, and the analyses are handled as a set of strings as long as they cannot be reduced to a single analysis. The notion of set of strings covers the two dimensions that linguists call the syntagmatic axis, i.e. that of the chronological sequence of elements in a given utterance, and the paradigmatic axis, i.e. the “or” relation between linguistic forms that can substitute for one another.

The connection between language processing and combinatorics on words is natural. Historically, linguists actually played a part in the beginning of the construction of theoretical combinatorics on words. Some of the terms in current use originate from linguistics: word, prefix, suffix, grammar, syntactic monoid... However, interpenetration between the formal world of computer theory and the intuitive world of linguistics is still a love story with ups and downs. We will encounter in this chapter, for example, terms that specialists of language processing use without bothering about what they mean in mathematics or in linguistics.

This chapter is organized around the main levels of any language modeling: first, how words are made from letters; second, how sentences are made from words. We will survey the basic operations of interest for language processing, and for each type of operation we will examine the formal notions and tools involved.

3.1. From letters to words

All the operations in the world between letters and words can be collectively denoted by the term “lexical analysis”. Such operations mainly involve finite automata and transducers. Specialists in language processing usually refer to these formal tools with the term “finite-state” tools, because they have a finite number of states.

3.1.1. Normalization of encoding

The computer encoding of the 26 letters of the Latin alphabet is fairly standardized. However, almost all languages need additional characters for their writing. European languages use letters with diacritics: accents (*é, è*), cedilla (*ç*), tilde (*ñ*), umlaut (*ü*)... There are a few ligatures, the use of some of them being standard in some conditions: *æ*, *œ*, *ß*, others are optional variants: *ff*, *fl*. The encoding of these extensions of 7-bit ASCII is by no means normalized: constructors of computers and software editors have always tended to propose divergent encodings in order to hold users captive and so faithful. Thus, *é* is encoded as 82 and 8E in two common extended ASCII codes, as 00E9 in UCS-2 Unicode, as C3A9 in UTF-8 Unicode, and named "é" by ISO 8879:1986 standard. The situation of other alphabets (Greek, Cyrillic, Korean, Japanese...) is similar. The encoding systems for the Korean national writing system are based on different levels: in KSC 5601-1992, each symbol represents a syllable; in "n-byte" encodings, each symbol represents a segment of a syllable, often a phoneme.

Thus, generally speaking, when an encoding is transliterated into another, a symbol may be mapped to a sequence of several symbols, or the reverse. Transliteration implies (i) cutting up input text into a concatenation of segments, and (ii) translating each segment. Both aspects depend on input and output encodings.

Transliteration is simple whenever it is unambiguous, i.e. when source encoding and target encoding convey exactly the same information in two different forms. The underlying formal objects are very simple. The set of possible segments in input text is a finite code (the input code). It is often even a prefix code, i.e. no segment is a prefix of another. Here is an example of an input code that is not prefix: consider transliterating a phoneme-based Korean encoding into a syllable-based encoding. A 5-symbol input sequence *kilto* must be segmented as *kil/to* in order to be translated into a 2-symbol output sequence, but *kilo* must be segmented as *ki/lo*.

In any case, encodings are designed so that transliteration can be performed by a sequential transducer.

For the reader's convenience, we will recall a few of the definitions of section 1.5. A finite transducer over the alphabets A, B is a finite automaton in which all edges have an input label $u \in A^*$ and an output label $v \in B^*$. The input alphabet A can be different from the output alphabet B, but they frequently have a nonempty common subset. The notation we will use is convenient when a transducer is considered as an automaton over a finite alphabet of the form $X \subset A^\star \times B^\star$, as in section 3.1.5, and when we define a formal notion of alignment, as in section 3.1.7. Elements of X will be denoted $(u\!:\!v)$ or $\begin{pmatrix} u \\ v \end{pmatrix}$ as in Fig. 3.1; edges will be denoted $(p, u\!:\!v, q)$. The label of a successful path of a transducer consists of a pair of sequences $(w\!:\!x) \in A^\star \times B^\star$. Corresponding input and output sequences may be of different lengths in number of symbols, and some of the edges may have input and output labels of different lengths. A transducer over A and B is input-wise literal if and only if all input labels are in $A|\varepsilon$, and input-wise synchronous if and only if they are in A. The set of labels of successful paths of a transducer is the transduction realized by the transducer. A transduction over A and B is a relation between $A^\star$ and $B^\star$. A transduction over A and B can be specified by a regular expression in the monoid $A^\star \times B^\star$ if and only if it is realized by a finite transducer.

A sequential transducer is a finite transducer with additional output labels attached to the initial and terminal states, and with the following properties:

- it has at most one initial state,
- it is input-wise synchronous,
- for each state p and input label $a \subset A$, there is at most one edge $(p, a : u, q) \in E$.

The output string for a given input string is obtained by concatenating the initial output label, the output label of the path defined by the input string, and the terminal output label attached to the terminal state that ends the path. With a sequential transducer, input sequences can be mapped into output sequences through input-wise deterministic traversal. All transductions realized by sequential transducers are word functions. Sequential transducers can be minimized (cf. section 1.5.2).

In practice, the output labels attached to terminal states are necessary for transliteration when input code is not prefix. The second and third properties above are obtained by adapting the alignment between input labels and output labels, i.e. by making them shorter or longer and by shifting parts of labels between adjacent edges. Fig. 3.1 shows a sequential transducer that transliterates *é* and *è* from their ISO 8879 names, "é" and "è", to their codes in an extended ASCII encoding, 82 and 8A.

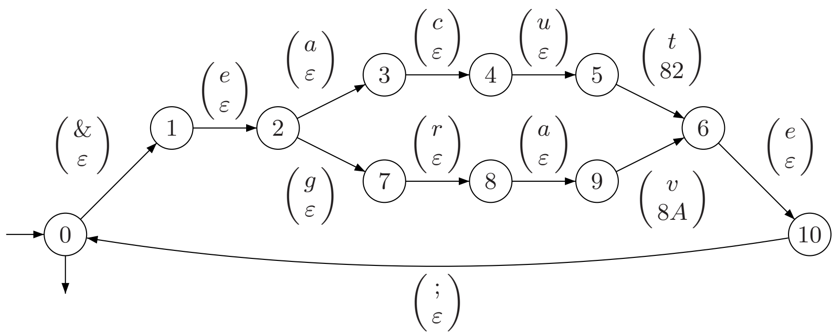


Figure 3.1. A sequential transducer that substitutes "82" for "é" and "8A" for "è".

The number of edges of transducers for normalization of character encoding is of the same order of magnitude as the sum of the lengths of the elements of the input code, say 30 if only letters are involved and 3000 if syllables are involved.

Transliteration from one encoding to another is ambiguous when the target system is more informative than the source system. For example, 7-bit ASCII encoding, frequently used in informal communication, does not make any difference between *e* and *é*, or between *oe* and the ligature *œ*. In a more elaborate encoding, these forms are not equivalent: *œ* is not a free variant for *oe*; it can be used in *cœur* but not in *coexiste*. Transliteration from 7-bit ASCII to an extended ASCII encoding involves recognizing more complex linguistic elements, like words. It cannot be performed by small sequential transducers.

The situation is even more complex in Korean and Japanese. In these languages, text can be entirely written in national writing systems, but Chinese characters are traditionally substituted for part of it, according to specific rules. In Japan, the use of Chinese characters in written text is standard in formal communication; in Korea, this traditional substitution is not encouraged by the authorities and is on the waning. Let us consider text with and without Chinese characters as two encodings. The version with Chinese characters is usually more informative than the one without: when a word element is ambiguous, it may have several transcriptions in Chinese characters, according to its respective meanings. However, the reverse also happens. For instance, an ambiguous Chinese character that evokes "music", "pleasure" or "love" in Korean words is pronounced differently, and transcribed *ak*, *lak*, *nak* or *yo* in the national writing system, depending on the words in which it occurs.

3.1.2. Tokenization

The first step in the processing of written text is helped by the fact that words are delimited by spaces. During Antiquity, this feature was exclusive to unvowelled script of Semitic languages; it developed in Europe progressively during the early Middle Ages (Saenger, 1997) and is now shared by numerous languages in the world.

Due to word delimitation, a simple computer program can segment written text into a sequence of words without recognizing them, e.g. without a dictionary. This process is called tokenization. Once it has been performed, words become directly available for further operations: statistics, full text indexation, dictionary lookup...

The formal basis of delimiter-based tokenization is the unambiguous use of certain characters as delimiters.

The alphabet of letters, A, and the alphabet of delimiters, D, are disjoint. A text is a sequence of letters and delimiters. After tokenization, it is a sequence of tokens. Word tokens are maximal occurrences of elements of $A^\star$ in the text. Delimiter tokens can be defined either as single delimiters:

Why/?/ /1/./ /Because/ /of/ /temperature/.

or as sequences of delimiters:

Why/? 1. /Because/ /of/ /temperature/.

Some symbols, like dash (-) and apostrophe (') in English, can be considered either as letters or as delimiters. In the first case, *trade-off* and *seven-dollar* are tokens; otherwise they are sequences of tokens. In any case, tokenization can be performed by simulating the two-state automaton of Fig. 3.2, and by registering a new word token whenever control shifts from state 1 to state 0.

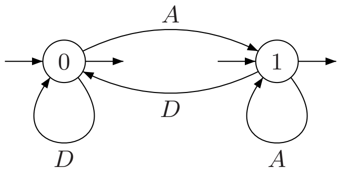


Figure 3.2. An automaton for written text tokenization.

In this section, we used the term "word" in its everyday sense; I would even say in its visual sense: a word in written text is something visibly separated by spaces. However, this naive notion of word does not always give the best results if we base further processing on it, because visual words do not always behave as units conveying a meaning. For example *white* does in *white trousers*, but not in *white wine*. We will return to this matter in section 3.1.4.

Delimiter-based tokenization is not applicable to languages written without delimitation between words, like Arabic, Chinese or Japanese. In these languages, written text cannot be segmented into words without recognizing the words. The problem is exactly the same with spoken text: words are not audibly delimited.

However, in some cases, another type of tokenization consists in identifying all the positions in the text where words are liable to begin. These positions cut up text into tokens. After that, words can be recognized as certain sequences of tokens. For instance, in Thai language, words can only begin and end at syllable boundaries, and syllable boundaries cannot be preceded or followed by any patterns of phonemes. These patterns can be recognized by a transducer.

3.1.3. Zipf's law

During the tokenization of a text or of a collection of texts, it is easy to build the list of all the different tokens in the text, to count the occurrences of each different token, and to rank them by decreasing number of occurrences. What is the relation between rank r and number of occurrences n_r? Zipf observed that the following law is approximately true:

$$n_r = n_1/r^a \qquad (3.1.1)$$

with $a \approx 1$. As a matter of fact, there are few frequent tokens, and many infrequent tokens. In experiments on French text, 1 token out of 2 was found to belong to the most frequent 139 tokens. In fact, for $20 \leq r \leq 2000$, n_r is a little higher than predicted by (3.1.1).

Several equations can be derived from Zipf's law. The number r_n of different tokens that occur at least n times is such that $n = n_1/r_n^a$, so:

$$r_n = \left(\frac{n_1}{n}\right)^{1/a}$$

The number of different tokens that occur between n and $n+1$ times is:

$$\left(\frac{n_1}{n}\right)^{1/a} - \left(\frac{n_1}{n+1}\right)^{1/a} \tag{3.1.2}$$

For large values of n and $a = 1$, this is approximately n_1/n^2, which is confirmed experimentally.

According to (3.1.2), the number of tokens that occur once (hapaxes) is proportional to $n_1^{1/a}$. It is easy to observe that the number of occurrences of a very frequent token is approximately proportional to the size of the text, i.e. n_1/N depends on the language but not on the text. This means that all texts comprise roughly the same proportion of hapaxes.

Can Zipf's law be used to predict the relation between the size of a text and the size of its vocabulary? The size of the text is the total number of occurrences of tokens,

$$N = n_1 + n_2 + \ldots + n_R$$

where R is the size of the vocabulary, i.e. the number of different tokens. With $a = 1$, we have:

$$N = n_1 \sum_{r=1}^{R} 1/r \approx n_1 \ln R$$

However, the relation between N and n_1 in this equation is not confirmed experimentally. Firstly, n_1 is proportional to N. Secondly, the growth of R with respect to N tends to slow down, because of the tokens that occur again, whereas this equation implies that it would speed up. Thirdly, if this law were accurate, R would grow unbounded with N, which means that the vocabulary of a language would be infinite. What is surprising and counter-intuitive is that a steady growth of R with respect to N is maintained for texts up to several million different tokens.

In other words, Zipf's law correctly predicts that a collection of texts needs to be very large and diverse to encompass the complete vocabulary of a language, because new texts will contain new words for a very long time. Experience shows, for example, that the proportion of vocabulary which is shared by one year's production of a newspaper and another year's production is smaller than simple intuition would suggest.

3.1.4. Dictionary compression and lookup

Most operations on text require information about words: their translation into another language, for example. Since such information cannot in general be computed from the form of words, it is stored in large databases, in association with the words. Information about words must be formal, precise, systematic and explicit, so that it can be exploited for language processing. Such information is encoded into word tags or lexical tags. Examples of word tags are given in Fig. 3.3. The tags in this figure record only essential information:

fit	*fit*	*A*
fit	*fit*	*N:s*
fit	*fit*	*V:W:P1s:P2s:P1p:P2p:P3p*
fitter	*fit*	*A:C*
fitting	*fit*	*V:G*
hop	*hop*	*N:s*
hop	*hop*	*V:W:P1s:P2s:P1p:P2p:P3p*
hope	*hope*	*N:s*
hope	*hope*	*V:W:P1s:P2s:P1p:P2p:P3p*
hoping	*hope*	*V:G*
hopping	*hop*	*V:G*
hot	*hot*	*A*
hot air	*hot air*	*N:s*
hotter	*hot*	*A:C*
open	*open*	*A*
open	*open*	*N:S*
open	*open*	*V:W:P1s:P2s:P1p:P2p:P3p*
open air	*open air*	*N:S*

Figure 3.3. The word tags for a few English words.

- the lemma, which is the corresponding form with default inflectional features, e.g. the infinitive, in the case of verbs,
- the part of speech: *A*, *N*, *V*...,
- the inflectional features.

Lemmas are necessary for nearly all applications, because they are indexes to properties of words. If all the vocabulary is taken into account, the tag set used in Fig. 3.3 has many thousands of elements, due to lemmas. Size of tag sets is a measure of the informative content of tags.

The operation of assigning tags to words in a text is called lexical tagging. It is one of the main objectives of lexical analysis. The reverse operation is useful in text generation: words are first generated in the form of lexical tags, then you have to spell them. In many languages, it is feasible to construct a list of roughly all words that can occur in texts. Such a list, with unambiguous word tags, is called an electronic dictionary[2], or a dictionary. The strange term "full-form dictionary" is also in use. An electronic dictionary is in the order of a million words. Such a list is always an approximation, due to the fact that new words continuously come into use: proper nouns, foreign borrowings, new derivatives of existing words...

[2] The term "electronic dictionary" emphasizes the fact that entries are designed for programs, whereas the content of "conventional dictionaries" is meant for human readers, no matter whether they are stored on paper or on electronic support.

In inflectional languages like English, the construction of an electronic dictionary involves generating inflected forms, like conjugated verbs or plurals. This operation is usually carried out with tables of suffixes, prefixes or infixes, or with equivalent devices.

What is considered as a word is not always clear, because words sometimes appear as combinations of words, e.g. *hot air* “meaningless talk”, *open air* “outdoors space”, *white wine*, which are called compound words. The situation is less clear with numerals, e.g. *sixty-nine*: linguistically, each of them is equivalent to a determiner, which is a word; technically, if we include them in the dictionary, they are another million words; syntactically, they are made of elements combined according to rules, but these rules are entirely specific to numerals and are not found anywhere in the syntax of the language. The status of such forms and of other examples like dates is not easy to assign. If they are considered as words, then the simplest form of description for them is a finite automaton. We will refer to such automata in section 3.2.2 by the term “local grammars”.

The most repetitive operation on an electronic dictionary is lookup. The input of this operation is word forms, and the output, word tags. Natural and efficient data structures for them are tries, with output associated to leaves, and transducers. In both cases, lookup is done in linear time with respect to the length of the word, and does not depend on the size of the dictionary.

Consider representing the dictionary in the form of a transducer. The dictionary is viewed as a finite set of word form/word tag pairs, i.e. a transduction. Alignment between input and output is based on the similarity between word forms and the lemmas included in word tags. This transduction is not a word function, since many word forms in a dictionary are associated with several word tags, like *fit* in Fig. 3.3:

The shoes are fit for travel
Max had a fit of fever
These shoes fit me

Due to this universal phenomenon, known as lexical ambiguity or homography, the transduction cannot be represented by a sequential transducer. A p-sequential transducer is a generalization of sequential transducers with at most p terminal output strings at each terminal state. A p-sequential transducer for the words in Fig. 3.3 is shown in Fig. 3.4. In this transducer, the symbol # stands for a space character. The notion of p-sequential transducer allows for representing a transduction that is not a word function without resorting to an ambiguous transducer. A transducer is ambiguous if and only if it has distinct paths with the same input label. In a p-sequential transducer, there are no distinct paths with the same input label; any difference between output labels of the same path must occur in terminal output strings.

In order to make the transducer p-sequential, lexically ambiguous word forms must be processed in a specific way: any difference between the several word tags for such a word form must be postponed to terminal output strings, by shifting parts of labels to adjacent edges. This operation may change the natural alignment between input and output, and increase the number of states and

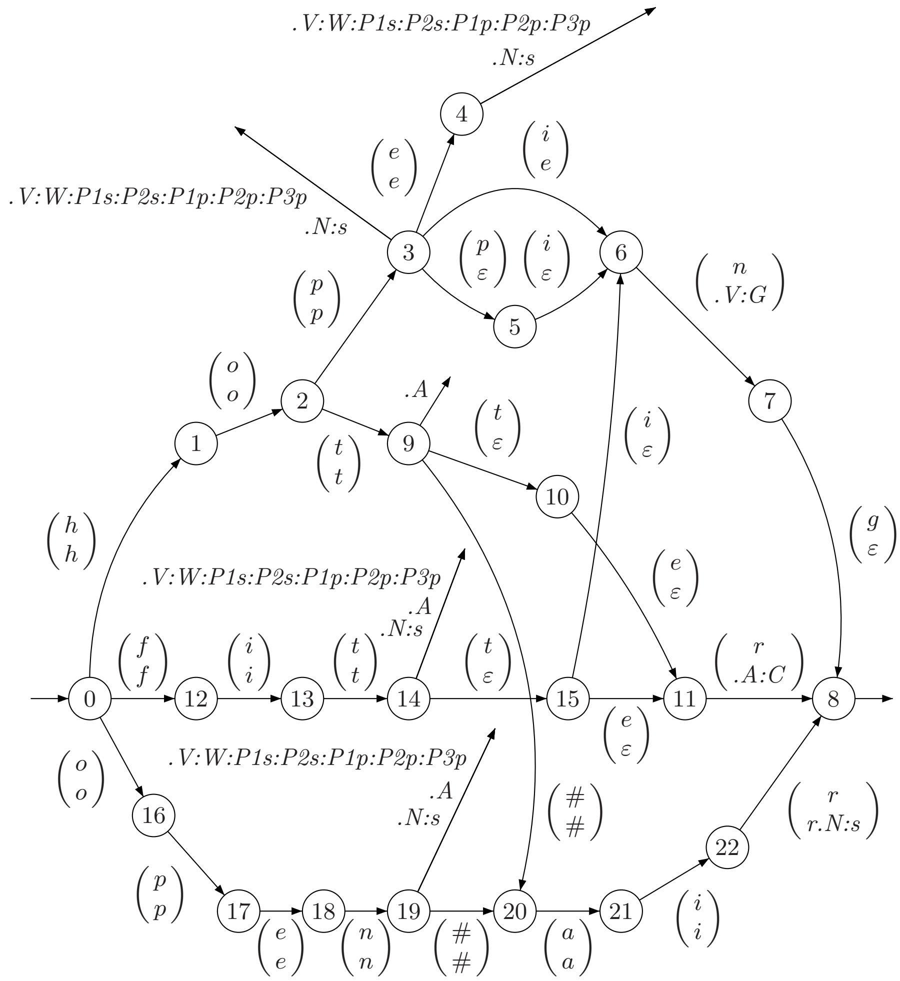


Figure 3.4. A p-sequential transducer for the words and tags in Fig. 3.3.

edges of the transducer, but the increase in size remains within reasonable proportions because inflectional suffixes are usually short. After this operation, a variant of algorithm ToSequentialTransducer (section 1.5) can be applied.

A dictionary represented as a transducer can be used to produce a dictionary for generation, by swapping input and output. The resulting transducer can be processed so as it becomes p-sequential too, provided that the dictionary is finite.

Fig. 3.5 shows an approximation of the preceding transducer by an acyclic automaton or DAWG. Most of the letters in the word form are identical to letters in the lemma and are not explicitly repeated in the output. The end of the output is shifted to the right and attached to terminal states, with an integer indicating how many letters at the end of the word form are not part of the lemma. When several output strings are possible for the same word, they are concatenated and the result is attached to a terminal state. During minimization of the DAWG, terminal states can be merged only if the output strings attached to them are identical. For the tag set used in Fig. 3.3, and for all the vocabulary, there are only about 2000 different output strings. The practical advantage of this solution is that output strings are stored in a table that need not be compressed and is easy to search for word tags.

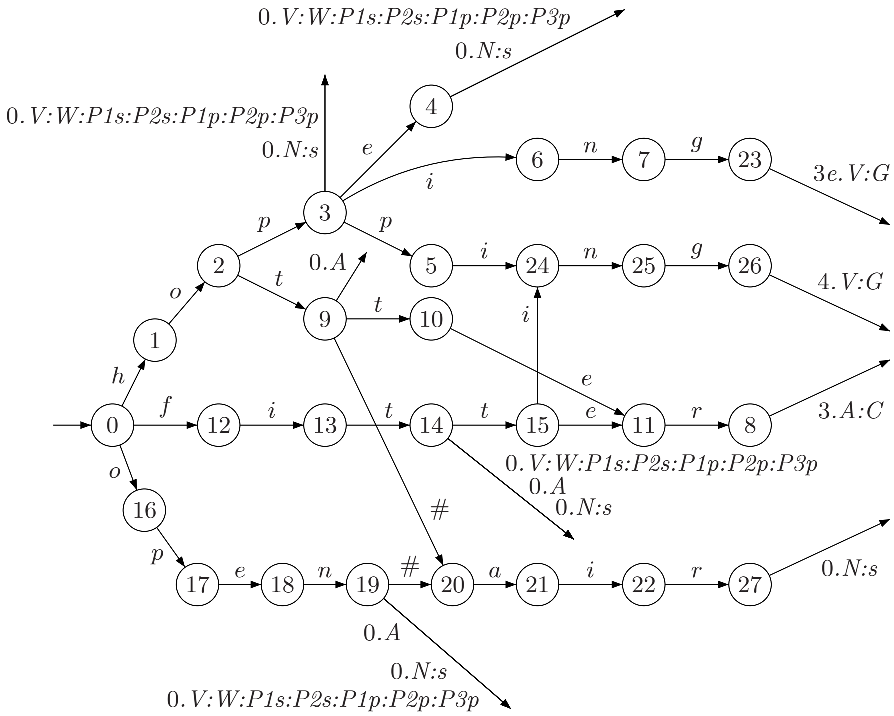


Figure 3.5. The DAWG for the words and tags in Fig. 3.3.

In the previous figures, we have presented the same dictionary in different forms. The form containing most redundancy is the list (Fig. 3.3): parts of words are repeated, not only in lemmas and inflected forms, but also across different entries. The DAWG (Fig. 3.5) is virtually free of this redundancy, but it is unreadable and cannot be updated directly. In fact, linguistic maintenance must be carried out on yet another form, the dictionary of lemmas used to generate the list of Fig. 3.3. The dictionary of lemmas is readable and presents little redundancy, two fundamental features for linguistic maintenance. But the only way to exploit it computationally is to generate the list – a form with huge redundancy – and then the DAWG. The flexibility of finite automata is essential to this practical organization.

The main difficulties with dictionary-based lexical tagging are lexical lacunae, errors and ambiguity.

Lexical lacunae, i.e. words not found in a dictionary, are practically impossible to avoid due to the continuous creation and borrowing of new words. Simple stopgaps are applicable by taking into account the form of words: for example, in English, a capitalized token not found in the dictionary is often a proper noun.

Lexical errors are errors producing forms which do not belong to the vocabulary of the language, e.g. *coronre* for *coroner*[3]. Lexical errors are impossible to distinguish from lexical lacunae. A few frequent errors can be inserted in dictionaries, but text writers are so creative that this solution cannot be implemented systematically. In order to deal with errors (find suggestions for corrections, retrieve lexical information about correct forms), an electronic dictionary can be used. By looking up in an error-tolerant way, we find correct forms that are close to the erroneous form.

Lexical ambiguity refers to the fact that many words should be assigned distinct tags in relation to context, like *fit*. About half the forms in a text are lexically ambiguous. Lexical ambiguity resolution is dealt with in section 3.2.4.

In some languages, sequences of words are written without delimiter in certain conditions, even if the sequence is not frozen. In German, *ausschwimmen* "to swim out" is the concatenation of *aus* "out" and *schwimmen* "swim". Obviously, dictionary lookup has to take a special form in cases where a token comprises several words.

Performing the lexical analysis of a text with a set of dictionaries requires adapted software, like the open-source system Unitex. Fig. 3.6 shows the result of the lexical analysis of an English text by Unitex. This system can also be used for the management of the dictionaries in their different forms, and for the operations on words that we will present in section 3.2.

3.1.5. Morphological analysis

Given a word in a written text, represented by a sequence of letters, how do you analyse it into a sequence of underlying morphological elements? This problem is conveniently solved by the dictionary methods of the preceding section, except when the number of morphological elements that make up words is too large. This happens with agglutinative languages. English and other Indo-European languages are categorized as inflected languages. A few agglutinative languages are spoken in Europe: Turkish, Hungarian, Finnish, Basque... and many others are from all other continents. In such languages, a word is a concatenation of morphological elements, usually written without delimiters[4]. For

[3] Errors can also produce words which belong to the vocabulary, like *corner*.

[4] When morphological elements are delimited by spaces, like in Sepedi, an African agglutinative language, the problem of recognizing their combinations is quite different.

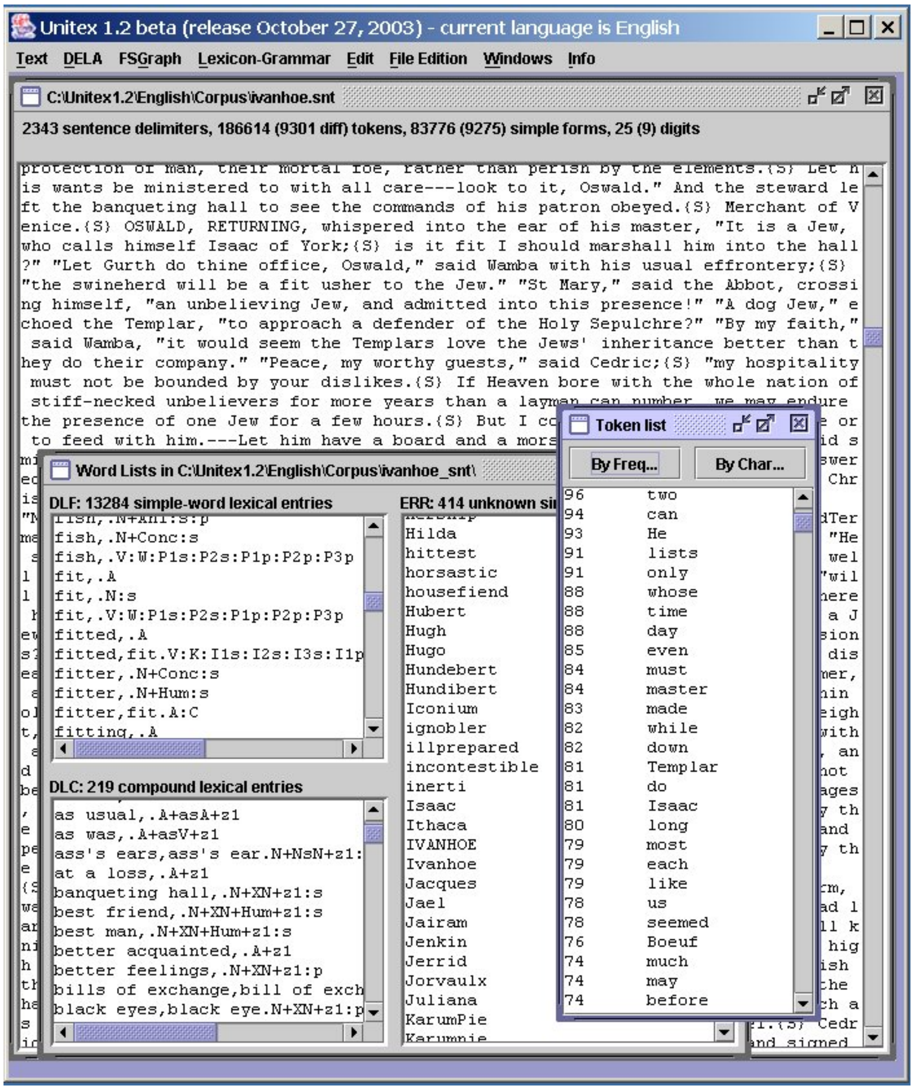


Figure 3.6. Lexical analysis of an English text by Unitex.

example, the following Korean sequence, transliterated into the Latin alphabet: *manasiôs'takojocha* "even that (he) met", comprises 6 elements:

- *mana* "meet"
- *si* (honorification of grammatical subject)
- *ôs'* (past)
- *ta* (declarative)

- *ko* "that"
- *jocha* "even"

and can be used in a sentence meaning "(The Professor) even (thought) that (he) met (her yesterday)". The form of each element can depend on its neighbors, so each element has a canonical form or lemma and morphological variants. There are two types of morphological elements: stems, which are lexical entries, like "meet" in the Korean example, and grammatical affixes, like tense, mood or case markers. Morphological analysis consists of segmenting the word and finding the lemma and grammatical tag of each underlying morphological element. The converse problem, morphological generation, is relevant to machine translation in case of an agglutinative target language: words are constructed as sequences of morphological elements, but you have to apply rules to spell the resulting word correctly.

Finite transducers are usually convenient for representing the linguistic data required for carrying out morphological analysis and generation. For example, Fig. 3.7 represents a part of English morphology as if it were agglutinative. This transducer analyses *removably* as the combination of three morphological elements, *remove.V*, *able.A* and *ly.ADV*, and inserts plus signs in order to delimit them. The transducer roughly respects a natural alignment between written forms and underlying analyses. It specifies two types of information: how written forms differ from underlying forms, and which combinations of morphological elements are possible. Grammatical codes are assigned to morphological elements: verb, adjective, tense/mood suffix, adverb. Some other examples of words analyzed by this transducer are *remove*, *removable*, *removed*, *removing*, *accept*, *acceptable*, *acceptably*, *accepted*, *accepting*, *emphatic*, *emphatically*, *famous* and *famously*. The four initial states should be connected to parts of the dictionary representing the stems that accept the suffixes represented in the transducer.

In this toy example, it would have been simpler to make a list of all suffixed forms with their tags. However, combinations of morphological elements are more numerous and more regular in agglutinative languages than in English, and they justify the use of a transducer.

Transducers of this kind obviously have to be manually constructed by linguists, which implies the use of a convenient, readable graphic form, so that errors are easily detected and maintenance is possible. A widely used set of conventions consists in attaching labels to states and not to edges. States are not explicitly numbered. This graphic form is sometimes called a "graph". For example, Fig. 3.8 shows the same transducer as Fig. 3.7 but with this presentation. The expressive power is the same. When the transducer is used in an operation on text or with another transducer, it is compiled into the more traditional form. During this compilation, states are assigned arbitrary numbers.

The main challenge with algorithmic tools for morphological processing is the need to observe two constraints: manually constructed data must be presented in a readable form, whereas data directly used to process text must be coded in adapted data structures. When no format is simultaneously readable and adapted to efficient processing, the data in the readable form must be auto-

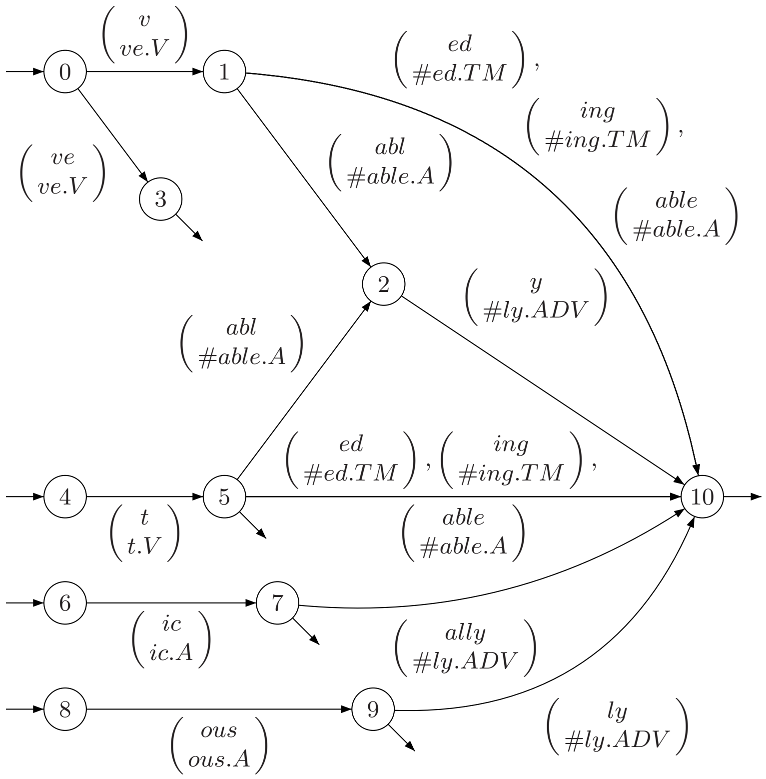


Figure 3.7. Morphological analysis in English.

matically compiled into the operation-oriented form. This organization should not be given up as soon as operation-oriented data are available: linguistic maintenance, i.e. correction of errors, inclusion of new words, selection of vocabulary for applications etc., can only be done in the readable form.

Transducers for morphological analysis are usually ambiguous. This happens when a written word has several morphological analyses, like *flatter*, analyzable as *flatter.V* in *Advertisements flatter consumers*; and as *flat.A+er.C* in *The ground is flatter here*. The fact that transducers are ambiguous is not a problem for linguistic description, since ambiguous transducers are as readable as unambiguous ones. However, it can raise algorithmic problems: in general, an ambiguous transducer cannot be traversed in an input-wise deterministic way. In inflected languages, this problem is avoided by substituting p-sequential transducers to ambiguous transducers, but this solution is no longer valid for most agglutinative languages. When ambiguity affects the first element in a long sequence of morphological elements, shifting output labels to terminal output strings would change the natural alignment between input and output to such

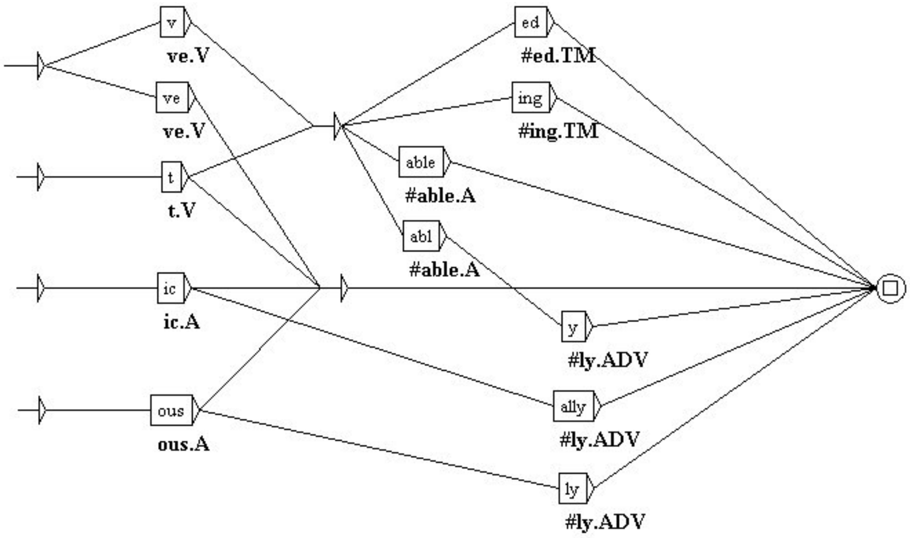


Figure 3.8. Morphological analysis in English.

an extent that the number of states and edges of the transducer would explode.

Therefore, algorithm ToSequentialTransducer is not applicable: ambiguous transducers have to be actually used. There are several ways of automatically reducing the degree of input-wise nondeterminism of an ambiguous transducer. We will see two methods which can be applied after the alignment of the transducer has been tuned so as to be input-wise synchronous (see section 3.1.1). Both methods will be exemplified on the transducer of Fig. 3.8, which has 4 initial states. These distinct initial states encode dependencies between stems and suffixes, as we will see in the last page of this section. For simplicity's sake, the stems are not included in this figure: thus, we will consider it as a collection of 4 transducers, and artificially maintain the 4 initial states.

The first method consists in determinizing (algorithm NFAtoDFA, section 1.3.3) and minimizing (section 1.3.4) the ambiguous transducer, considering it as an automaton over a finite alphabet $X \subset A^\star \times B^\star$. In general, the resulting transducer is still ambiguous: distinct edges can have the same origin, the same input label, and distinct ends, $(p, a : u, q)$ and $(p, a : v, r)$, but only if their output labels u and v are distinct. The transducer of Fig. 3.9 is the result of the application of this method to the transducer of Fig. 3.8. Applying the resulting transducer to a word involves a variant of the nondeterministic search of section 1.3.2 (algorithm IsAccepted), but the search is quicker than with the original transducer, because algorithm NFAtoDFA reduces the nondeterminism of the transducer.

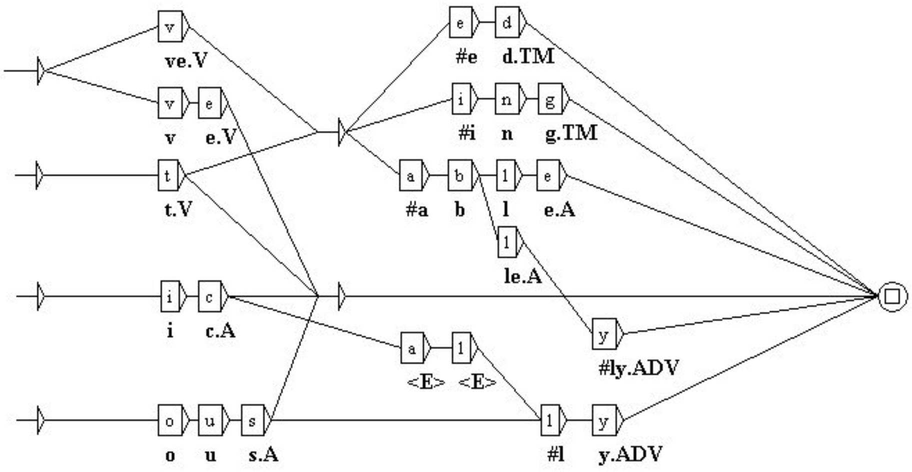


Figure 3.9. An ambiguous transducer determinized as an automaton.

In order to introduce the second method, we define a new generalization of p-sequential transducers. We will allow differences between output labels of the same path to occur at any place as long as they remain strictly local. Formally, a generalized sequential transducer is a finite transducer with a finite set of output labels $I(i)$ for the initial state i, a finite set of output labels $T(q)$ for each terminal state q, and with the following properties:

- it has at most one initial state,
- it is input-wise synchronous,
- for each pair of edges $(p, a\!:\!u, q), (p, a\!:\!v, r)$ with the same origin and the same input label, $q = r$.

A transduction is realized by a generalized sequential transducer if and only if it is the composition of a sequential transduction with a finite substitution. Thus, such a transduction is not necessarily a word function: two edges can have the same origin, the same input label, the same end and distinct output labels, $(p, a\!:\!u, q)$ and $(p, a\!:\!v, q)$. However, given the input label of a path, a generalized sequential transducer can be traversed in an input-wise deterministic way, even if it is ambiguous.

The second method constructs a generalized sequential transducer equivalent to the ambiguous transducer. When two edges with the same origin and the same input label have different output labels and different ends, output labels are shifted to adjacent edges to the right, but not necessarily until a terminal state is reached. The condition for ceasing shifting a set of output strings to the right is the following. Consider the set $E_{p,a}$ of all edges with origin p and input label a. Each edge $e \in E_{p,a}$ has an output label $u_e \in B^*$ and an end $q_e \in Q$. Consider the finite language $L_{p,a} \subset B^*Q$ over the alphabet $B \cup Q$ defined by $L_{p,a} = \{u_e q_e | e \in E_{p,a}\}$. If we can write $L_{p,a} = MN$ with $M \subset B^*$

and $N \subset B^*Q$, then

- create a new state r; let r be terminal if and only if at least one of the states q_e is terminal;
- substitute a new set of edges for $E_{p,a}$: the edges $(p, a\!:\!v, r)$ for all $v \in M$;
- shift the rest of output labels further to the right by replacing each edge $(q_e, b : w, s)$ with the edges $(r, b : xw, s)$ for all $x \in N$; for each terminal state among the states q_e, substitute $NT(q_e)$ for $T(q_e)$.

There can be several ways of writing $L_{p,a} = MN$: in such a case, the longer the elements of M, the better.

If the transduction realized by the ambiguous transducer is finite, this algorithm terminates; otherwise it is not certain to terminate. If it does, we obtain an equivalent generalized sequential transducer like that of Fig. 3.10.

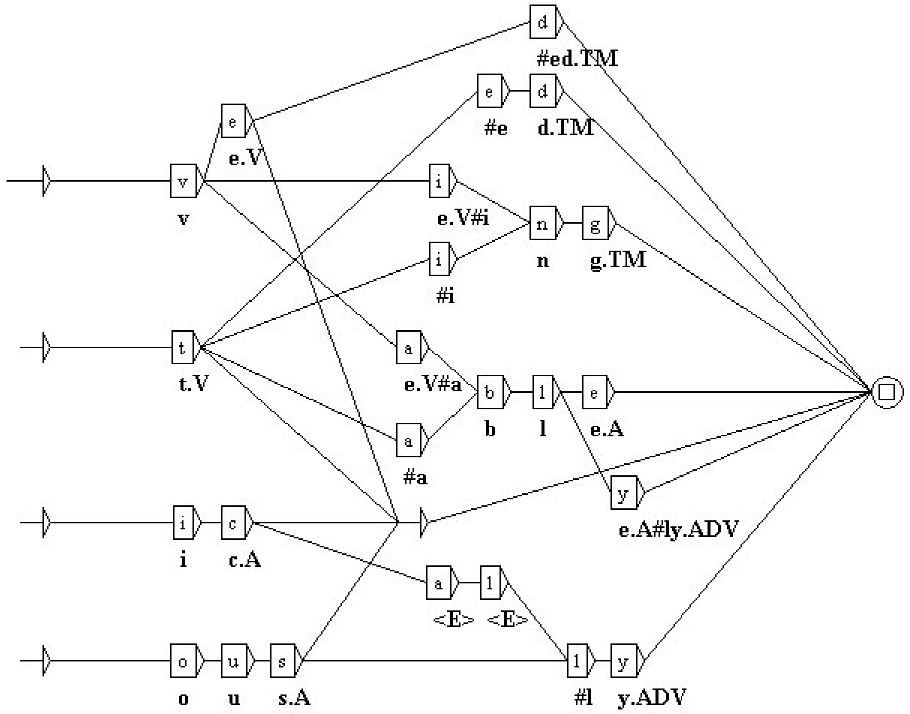


Figure 3.10. A generalized sequential transducer.

Transducers for morphological analysis like those of Fig. 3.7–3.10 can be used to produce transducers for morphological generation, by swapping input and output. The resulting transducer can be processed with the same methods as above in order to reduce nondeterminism.

When observable forms and underlying lemmas are very different, the description of morphology becomes complex. At the same time, it must still be hand-crafted by linguists, which requires that it is made of simple, readable parts, which are combined through some sort of compilation. For example, if both morphological variations and combinatorial constraints are complex, they are better described separately. Combinatorial constraints between morphological elements are described in an automaton at the underlying level, i.e. of lemmas and grammatical codes, as in Fig. 3.11.

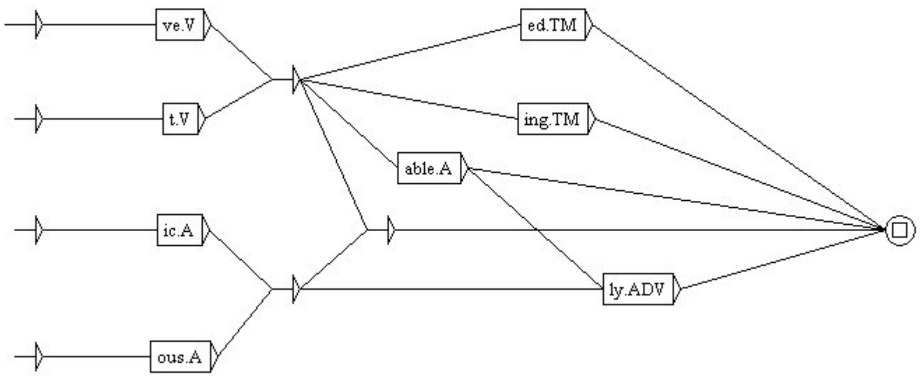


Figure 3.11. Combinatorial constraints between morphological elements.

Morphological changes are described in a transducer, with input at the level of written text and output at the underlying level. This is done in Fig. 3.12, which is more complex than Fig. 3.8, but also more general: it allows for more combinations of suffixes, i.e. *-ingly*, which was not included in Fig. 3.8 because it is not acceptable combined with *remove*.

How can we use these two graphs for morphological analysis? There are two solutions. The simpler solution applies the two graphs separately. When we apply the transducer of Fig. 3.12 to a word, we obtain, in general, an automaton. The automaton has several paths if several analyses are possible, as with *flatter*. Then when we compute the intersection of this automaton with that of Fig. 3.11, this operation selects those analyses that obey the combinatorial constraints. The algorithm of intersection of finite automata is based on the principle that the set of states of the resulting automaton is the Cartesian product of the sets of states of the input automata.

A more elaborate solution consists in performing part of the computation in advance. The automaton of Fig. 3.11 and the transducer of Fig. 3.12 do not depend on input text; they can be combined into the transducer of Fig. 3.8. If the automaton recognizes a set L and the transducer realizes a relation R, the operation consists in computing a transducer that realizes the relation R with its output restricted to L. This can be implemented, for instance, by applying algorithm ComposeTransducers (section 1.5) to the transducer of R and a transducer realizing the identity of L. Note that this algorithm is a variant of

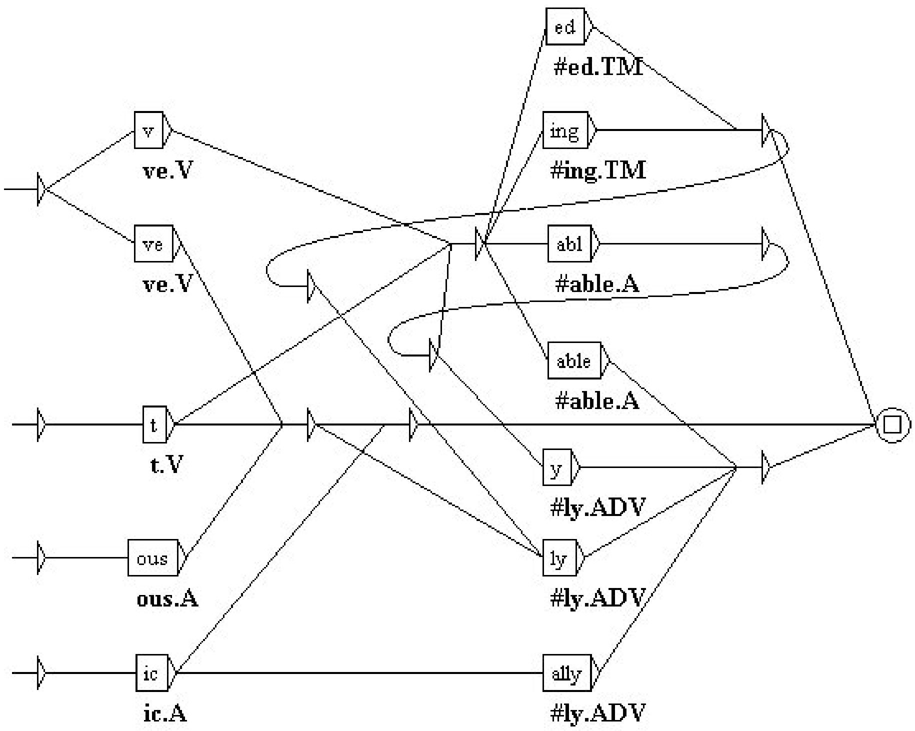


Figure 3.12. Morphological changes.

the algorithm of intersection of finite automata.

Morphological analysis and generation are not independent of the dictionary of stems: combinations of stems with affixes obey compatibility constraints, e.g. the verb *fit* does not combine with the suffix *-able*; stems undergo morphological variations, like *remove* in *removable*. Due to such dependencies, morphological analysis, in general, cannot be performed without vocabulary recognition. A dictionary of stems is manually created in the form of a list of many thousands of items and then compiled, so the interface with a transducer for morphological analysis requires practical organization. Combinatorial constraints between stems and affixes are represented by assigning marks to stems to indicate to which initial states of the automaton each stem must be connected. During compilation, the dictionary of stems and the automaton of combinatorial constraints are combined into an automaton. Morphological variations of stems are taken into account in the transducer; if analogous stems behave differently in an unpredictable way, like *fit/fitted* and *profit/profited*, marks are assigned to stems and the transducer refers to these marks in its output. If these provisions are taken, the operation on the automaton of constraints and the transducer of variations can be performed as above and produces a satisfactory result.

In this case, the description is distributed over two data sets: an automaton and a transducer, and the principle of the combination between them is that the automaton is interpreted as a restriction on the output part of the transducer.

It is often convenient to structure manual description in the form of more than two separate data sets: for example, one for the final *e* of verbs like *remove*, another for the final *e* of -*able*, another for variations between the forms -*ly*, -*ly*, -*y* of the adverbial suffix etc. This strategy can be implemented in three ways, depending on the formal principle adopted to combine the different elements of description: composition of transductions, intersection of transducers, and commutative product of bimachines.

3.1.6. Composition of transductions

The simplest of these three techniques involves the composition of transductions. Specialists in language processing usually refer to this operation by the bucolic term "cascade". The principle is simple. The data for morphological analysis or generation consists of a specification of a transduction between input strings and output strings. This transduction can be specified with several transducers. The first transducer is applied to input strings, the next transducer to the output of the first, and so on. The global transduction is defined as the composition of all the transductions realized by the respective transducers.

For example, Fig. 3.8 is equivalent to the composition of the transductions specified by Figs. 3.13–3.16. Fig. 3.13 delimits and tags morphological elements,

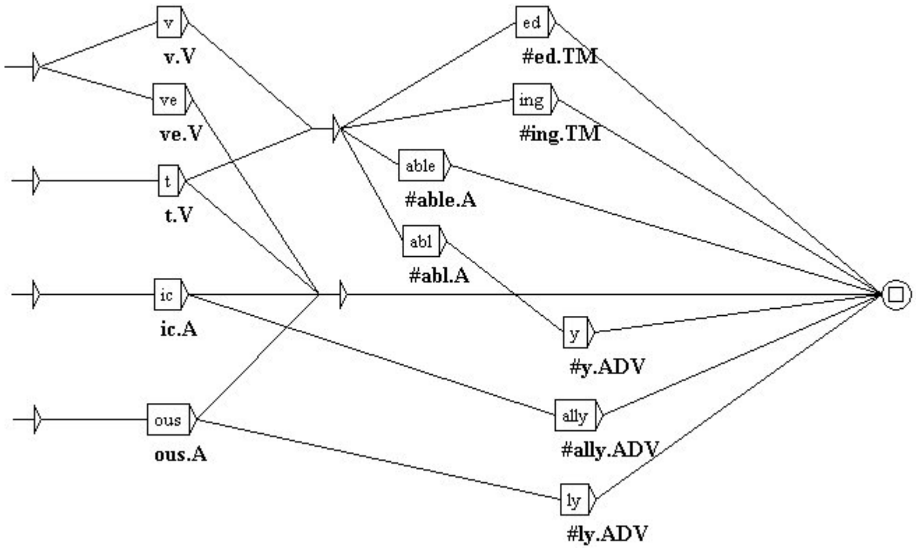


Figure 3.13. A cascade: first transducer.

but does not substitute canonical forms for variants. Fig. 3.14 inserts the final *e* of the canonical form of *remove*. In Fig. 3.14, the input label @ stands for

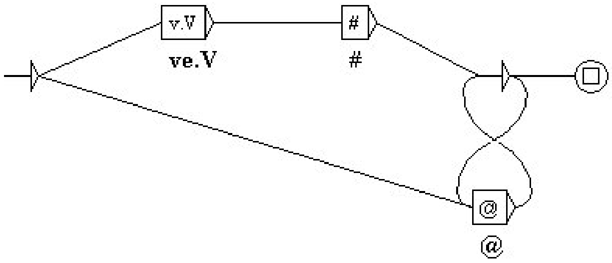


Figure 3.14. A cascade: second transducer.

a default input symbol: it matches the next input symbol if, at this point of the transducer, no other symbol matches. The output label @ means an output symbol identical to the corresponding input symbol. Fig. 3.15 inserts the final *e* of the canonical form of -*able*. Fig. 3.16 assigns the canonical form to the

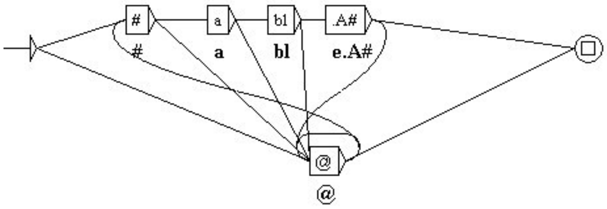


Figure 3.15. A cascade: third transducer.

variants of the adverbial suffix -*ly*.

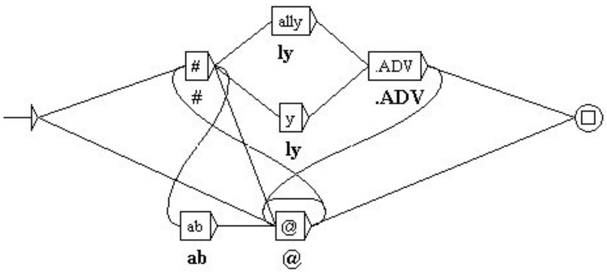


Figure 3.16. A cascade: fourth transducer.

During the application of a transducer, the input string is segmented according to the input labels of the transducer, and the output string is a concatenation of output labels. When transducers are applied as a cascade, the segmentation of the output string of a transducer is not necessarily identical to the segmentation induced by the application of the next. The global transduction is not changed if we modify the alignment of one of the transducers, provided that it realizes the same transduction.

As an alternative to applying several transducers in sequence, one can precompute an equivalent transducer by algorithm ComposeTransducers, but the application of the resulting transducer is not necessarily quicker, depending on the number, size and features of the original transducers.

The principle of composition of rules was implemented for the first time in... the 5th century B.C., in Panini's Sanskrit grammar, in order to define Sanskrit spelling, given that the form of each element depends on its neighbors.

Composition of relations is not a commutative operation. In our example of a cascade, the transductions of Figs. 3.14–3.16 can be permuted without changing the result of the composition, but they must be applied after Fig. 3.13, because they use the boundaries of morphological elements in their input, and these boundaries are inserted by the transduction of Fig. 3.13. In general, simple transductions read and write only in a few regions of a string, but interactions between different transductions are observed when they happen to read or write in the same region.

The principle of defining a few levels in a determined order between the global input level and the global output level is often natural and convenient. The alphabet of each intermediate level is a subset of $A \cup B$. In morphological generation, the level of underlying morphological elements may have something to do with a previous state of the language, the sequence of levels being connected to successive periods of time in the history of language changes.

However, in a language with complex morphological variations represented by dozens of rules, the exclusive use of composition involves dozens of ordered levels. This complicates the task of the linguist, because he has to form a mental image of each level and of their ordering.

Intuitively, when two morphological rules are sufficiently simple and unrelated, one feels that it should be possible to implement them independently, without even determining in which order they apply: hence the term "simultaneous combination". In spite of this intuition, rules cannot be formalized without specifying how they are interpreted in case of an overlap between the application sites of several rules (or even of the same rule): if rules apply to two sites uv and vw, the value of v taken into account for uvw can involve the input or the output level, or both. Various formal ways of combining formal rules have been investigated. Two main forms of simultaneous combination are presently in use.

3.1.7. Intersection of transducers

The intersection of finite transducers can be used to specify and implement morphological analysis and generation. The alignment between input and output strings is an essential element of this model. This alignment must be literal, i.e. each individual input or output symbol must be aligned either with a single symbol or with ε. Several alignments are usually acceptable, e.g.

$$\binom{u}{u}\binom{s}{s}\binom{\varepsilon}{e}\binom{\varepsilon}{.V}\binom{\varepsilon}{\#}\binom{e}{e}\binom{d}{d}\binom{\varepsilon}{.TM}$$

and

$$\binom{u}{u}\binom{s}{s}\binom{e}{e}\binom{\varepsilon}{.V}\binom{\varepsilon}{\#}\binom{\varepsilon}{e}\binom{d}{d}\binom{\varepsilon}{.TM}$$

but one must be chosen arbitrarily.

Formally, an alignment over A and B is a subset of the free monoid X^*, where X is a finite subset of $A^\star \times B^\star$. An alignment is literal if it is a subset of $((A \mid \varepsilon) \times (B \mid \varepsilon))^*$.

The alignment is determined in order to specify explicitly the set of all pairs $(u{:}v) \in (A \mid \varepsilon) \times (B \mid \varepsilon)$ that will be allowed in aligned input/output pairs for all words of the language. Since all elements in the alignment will be concatenations of elements in this set, we can call it X. In the English example above, this set can comprise letters copied to output:

$$\binom{v}{v},\binom{e}{e},\binom{d}{d},\binom{i}{i},\binom{n}{n},\binom{g}{g},\binom{a}{a},\binom{b}{b},$$

$$\binom{l}{l},\binom{t}{t},\binom{c}{c},\binom{y}{y},\binom{o}{o},\binom{u}{u},\binom{s}{s},$$

plus a few insertions:

$$\binom{\varepsilon}{e},\binom{\varepsilon}{.V},\binom{\varepsilon}{\#},\binom{\varepsilon}{.TM},\binom{\varepsilon}{.A},\binom{\varepsilon}{l},\binom{\varepsilon}{.ADV},$$

and two deletions of letters:

$$\binom{a}{\#},\binom{l}{\varepsilon}$$

The set of aligned input/output pairs for all words of the language is viewed as a language over the alphabet X. This language is specified as the intersection of several regular languages. Each of these languages expresses a constraint that all input/output pairs must obey, and the intersection of the languages is the set of pairs that obey simultaneously all the constraints. Since these regular languages share the same alphabet $X \subset A^* \times B^*$, they can be specified by transducers over A and B. For example, the transducers in Figs. 3.17–3.20 specify necessary conditions of occurrence for some of the elements of X. In Fig. 3.17, the label @ denotes a default symbol. It matches the next member of X if and only if no other label explicitly present at this point of the graph

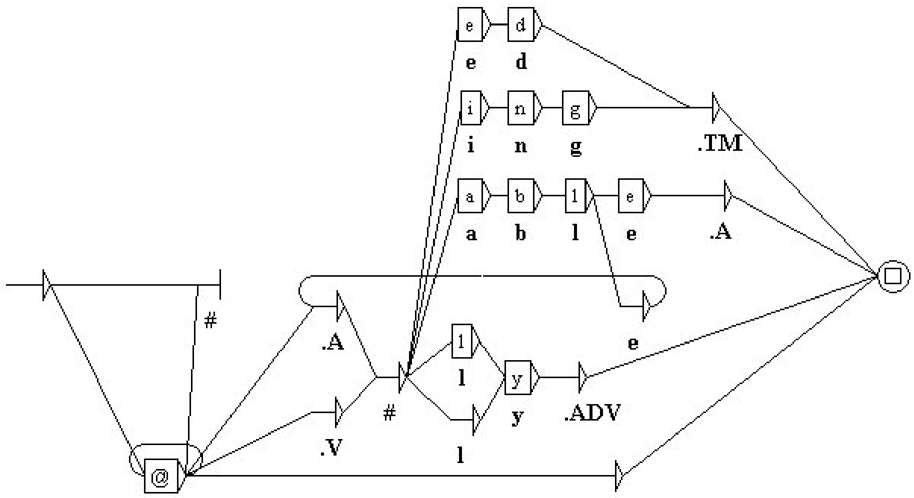


Figure 3.17. Conditions of occurrence of $(\varepsilon : \#)$.

does. One of the states has no outgoing edge and is not terminal: it is a sink state which is used to rule out the occurrence of $(\varepsilon : \#)$ when it is not preceded by $(\varepsilon : .A)$ or $(\varepsilon : .V)$.

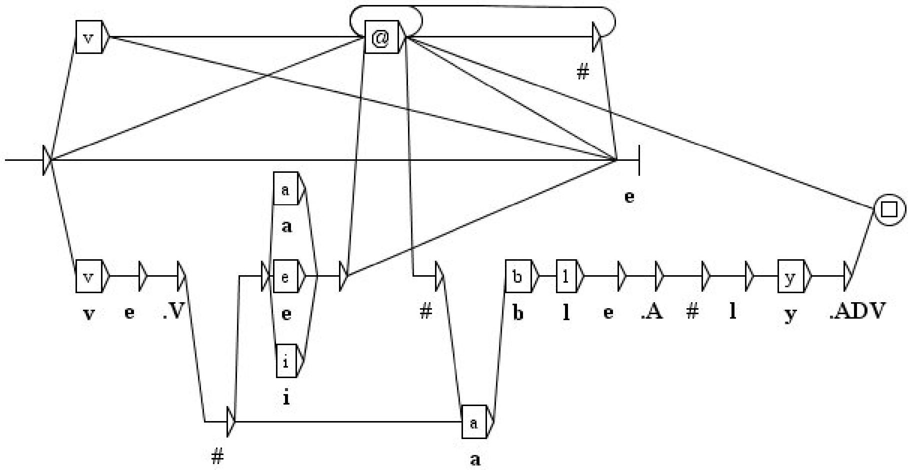


Figure 3.18. Conditions of occurrence of $(\varepsilon : e)$.

In order to be complete, we should add transducers to specify the conditions

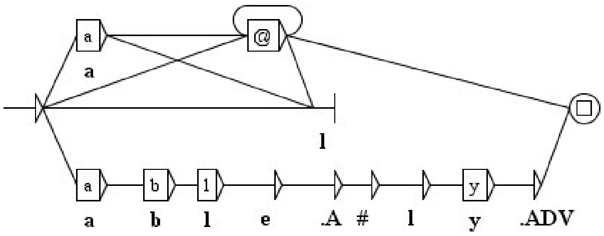


Figure 3.19. Conditions of occurrence of $(\varepsilon\!:\!l)$.

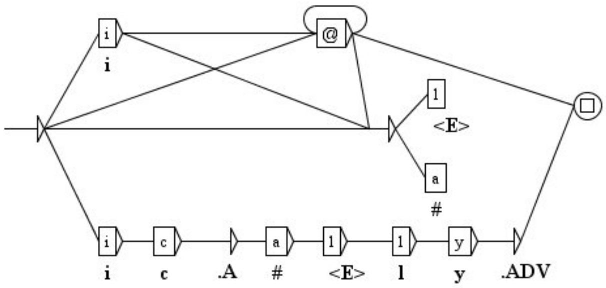


Figure 3.20. Conditions of occurrence of $(a\!:\!\#)$ and $(l\!:\!\varepsilon)$.

of occurrence of $(\varepsilon\!:\!.V)$, $(\varepsilon\!:\!.TM)$, $(\varepsilon\!:\!.A)$ and $(\varepsilon\!:\!.ADV)$.

The intersection of transducers is computed with the algorithm of intersection of automata, considering transducers as automata over X. The resulting transducer checks all the constraints simultaneously. This operation of intersection of transducers is equivalent to the intersection of languages in the free monoid $X^\star$, but not to the intersection of relations in $A^\star \times B^\star$, because the intersection of relations does not take into account alignment. (In addition, an intersection of regular relations is not necessarily regular.)

As opposed to the framework of composition of transductions, all the transducers describe correspondences between the same input level and the same output level. This is why this model is called "two-level morphology". Composition of transductions and intersection of transducers are orthogonal formalisms, and they can be combined: several batches of two-level rules are composed in a definite order.

Two-level constraints expressed as transducers are hardly readable, and expressing them as regular expressions over X would be even more difficult and error-prone. In order to solve this problem of readability, specialists in two-level morphology have designed an additional level of compilation. Rules are expressed in a special formalism and compiled into transducers. These transducers are then intersected together. The formalism of expression of two-level rules involves logical operations and regular expressions over X. For example, the following rule is equivalent to Fig. 3.17:

$$\binom{\varepsilon}{\#} \Longrightarrow \left(\binom{\varepsilon}{.A} \mid \binom{\varepsilon}{.V}\right) \underline{\quad\quad} \left(\binom{a}{a}\binom{b}{b}\binom{l}{l}\binom{\varepsilon}{e}\binom{\varepsilon}{.A}\right)^{*}$$
$$\left(\left(\left(\binom{e}{e}\binom{d}{d} \mid \binom{i}{i}\binom{n}{n}\binom{g}{g}\right)\binom{\varepsilon}{.TM} \mid \binom{a}{a}\binom{b}{b}\binom{l}{l}\binom{e}{e}\binom{\varepsilon}{.A} \mid \right.\right.$$
$$\left.\left.\left(\binom{l}{l} \mid \binom{\varepsilon}{l}\right)\binom{y}{y}\binom{\varepsilon}{.ADV}\right)\right)$$

This type of rule is more readable than a transducer, because it is structured in three separate parts: the symbol involved in the rule, here $(\varepsilon : \#)$, the left context (before $\underline{\quad\quad}$), and the right context.

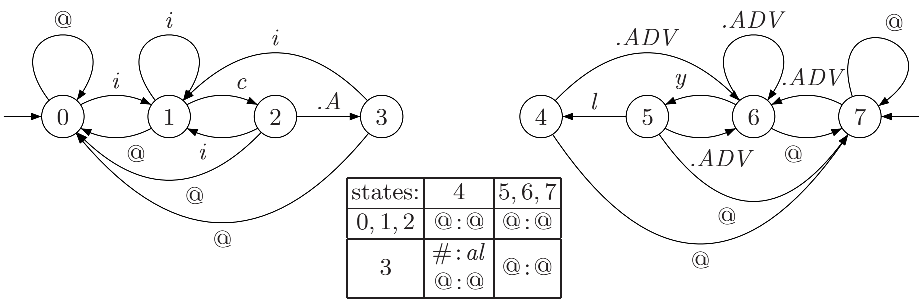
In this model, input and output are completely symmetrical: the same description is adapted for morphological analysis and generation.

3.1.8. Commutative product of bimachines

A bimachine is structured in three parts:

- a description of the left context required for the rule to apply,
- a similar description of the right context, and
- a mapping table that specifies a context-dependent mapping of input symbols to output symbols.

As opposed to two-level rules, left and right context are described only at input level. Fig. 3.21 is a representation of a bimachine that generates the variant *-ally* of the adverbial suffix *-ly* in *emphatically*.



states:	4	5, 6, 7
0, 1, 2	@ : @	@ : @
3	# : *al* @ : @	@ : @

Figure 3.21. Bimachine generating the variant *-ally* of the adverbial suffix *-ly*.

In this figure, the automaton on the left represents the left context and recognizes occurrences of the sequence $ic.A$. Whenever this sequence occurs, the automaton enters state 3. In the automaton, the label @ represents a default symbol: it matches the next input symbol if no other label at this point of the automaton does. The automaton on the right similarly recognizes occurrences of *ly.ADV*, but from right to left. Whenever this sequence occurs, the automaton enters state 4. The table specifies the mapping of input symbols to output symbols. The alphabets A and B have a nonempty common subset. In the table, @ : @ represents a default mapping: any input symbol not explicitly specified in the table is mapped onto itself. The symbol # is mapped to *al* when its left and right context is such that the respective automata are in states 3 and 4, i.e. when it is preceded by *ic.A* and followed by *ly.ADV*. Other symbols in such a context, and all symbols in other contexts, are copied to output. Thus, the bimachine maps occurrences of *ic.A#ly.ADV* to *ic.Aally.ADV* and leaves everything else unchanged. The input/output alignment that underlies the bimachine is always input-wise synchronous.

Formally, a bimachine over alphabets A and B is defined by

- two deterministic automata over A; let $\overrightarrow{Q}$ and $\overleftarrow{Q}$ be the sets of states of the two automata; the distinction between terminal vs. non-terminal states is not significant;
- a function $\gamma : \overrightarrow{Q} \times A \times \overleftarrow{Q} \longrightarrow B^*$, which is equivalent to the mapping table in Fig. 3.21.

The transduction realized by a bimachine is defined as follows. One performs a search in the left automaton controlled by the input word $u = u_1 u_2 \cdots u_n$. If this search is possible right until the end of the word, a sequence $\overrightarrow{q_0}\overrightarrow{q_1} \cdots \overrightarrow{q_n}$ of states of the left automaton is encountered, where $\overrightarrow{q_0}$ is the initial state. A similar search in the right automaton is controlled by $u_n \cdots u_2 u_1$. If the search can be completed too, states $\overleftarrow{q_n} \cdots \overleftarrow{q_1}\overleftarrow{q_0}$ of the right automaton are encountered, where $\overleftarrow{q_n}$ is the initial state.

The output string for the symbol u_i of u is $\gamma(\overrightarrow{q_{i-1}}, u_i, \overleftarrow{q_i})$ and the output for u is the concatenation of these output strings. If one of the searches could not be completed, or if one of the output strings for the letters is undefined, then the output for u is undefined.

A transduction is realized by a bimachine if and only if it is regular and a function.

The use of bimachines for specifying and implementing morphological analysis or generation requires that they can be combined to form complete descriptions. In the mapping table of Fig. 3.21, the default pair @ : @ occurs in all four cases; the bimachine specifies an output string for some occurrences of #, and copies all other occurrences of input symbols. We will say that the bimachine "applies" to these occurrences of #, and "does not apply" to other occurrences of input symbols. In morphology, separate rules belonging to the same description are complementary in so far as they do not "apply" to the same occurrences of input symbols. This idea can be used to define a notion of combination of bimachines over the same alphabets A and B.

Formally, we say that a bimachine "applies" to an input symbol a in a given context, represented by two states $\overrightarrow{q}$ and $\overleftarrow{q}$, if and only if $\gamma(\overrightarrow{q}, a, \overleftarrow{q})$ either is undefined or is not equal to a. It "does not apply" if and only if

$\gamma(\overrightarrow{q}, a, \overleftarrow{q}) = a$. If two bimachines never apply to the same symbol in the same input sequence, a new bimachine over the same input and output alphabets A and B can be defined so that the output for a given input symbol is specified by the bimachine that applies. The output is a copy of the input symbol if none of them applies. (Each automaton of the new bimachine is constructed from the corresponding automata of the two bimachines, with the algorithm of intersection of automata.) This operation on bimachines is commutative and associative; its neutral element is a bimachine that realizes the identity of A. We call this operation "commutative product".

The commutative product of a finite number of bimachines is defined if and only if it is defined for any two of them.

With this operation, linguists can manually construct separate bimachines, or rules, and combine them. These manually constructed rules must also be readable. This can be achieved by ensuring that the rules are presented according to the following conventions and have the following properties.

- Final states are specified in the two automata. The content of the mapping table does not depend on the particular states reached when exploring the context, but only on whether these states are terminal or not. For example, in Fig. 3.21, states 3 and 4 would be specified as terminal.
- In the mapping table, whenever at least one of the two states representing the context is non-terminal, input symbols are automatically copied to output, as in Fig. 3.21. When both states are terminal, only the input/output pairs for which the output string is different from the input symbol are specified. Let I be the set of input symbols that occur in the input part of these pairs: if both states are terminal and the input symbol is in I, the rule applies; otherwise, it does not apply and input is copied to output.
- The languages recognized by the two automata are of the form A^*L and A^*R, as in Fig. 3.21. Therefore, it suffices to specify L and R; automata for A^*L and A^*R can be automatically computed. In addition, the mirror image of R is specified instead of R itself, for the sake of readability.

The bimachine of Fig. 3.21 has these properties and is represented with these conventions in Fig. 3.22.

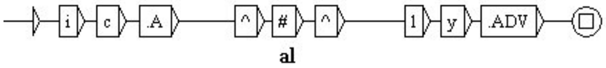


Figure 3.22. The bimachine of Fig. 3.21 with the conventions for manually constructed rules.

This figure represents L, R and the input/output pairs for which the rule applies. These three parts are separated by the states labeled $\wedge$.

The commutative product of two rules is defined if and only if $A^*L_1 \cap A^*L_2$, $A^*R_1 \cap A^*R_2$ and $I_1 \cap I_2$ are not simultaneously nonempty. This condition is tested automatically on all pairs in a set of rules written to be combined by commutative product. If the three intersections are simultaneously nonempty for a pair of rules, the linguist is provided with the set of left contexts, right contexts and input symbols for which the two rules conflict, and he/she can modify them in order to resolve the conflict. (A hierarchy or priorities between rules would theoretically be possible but would probably make the system more complex and its maintenance more difficult.)

The advantages of bimachines for specifying and implementing morphological analysis and generation are their readability and the fact that only differences between input and output need to be specified.

Bimachines are equivalent to regular word functions and, in principle, cannot represent ambiguous transitions. They have to be adapted in order to allow for limited variations in output. Take, for example, the generation of the preterite of *dream*: for a unique underlying form, *dream.V#ed.TM*, where *#ed.TM* is an underlying tense/mood suffix, there are two written variants: *dreamed* and *dreamt*. Such variations are limited; in agglutinative languages, they can occur at any point of a word, not necessarily just at the end. This problem is easily solved in the same way as we did for minimizing ambiguous transducers in section 3.1.5: by composition with finite substitutions. Bimachines realize transductions; several of these transductions can be composed in a definite order together or with finite substitutions.

In the example of *dream.V#ed.TM*, the two variants can be generated by introducing 3 new symbols *1*, *2* and *3*, and

- a bimachine that produces *dream.V#1ed.TM*,
- a finite substitution producing *dream.V#2ed.TM* and *dream.V#3ed.TM*, and
- a second bimachine that outputs *dreamed* for *dream.V#2ed.TM* and the variant *dreamt* for *dream.V#3ed.TM*.

However, a bimachine is an essentially deterministic formalism. It is adequate for the direct description of morphological generation, because the underlying level is more informative and less ambiguous than the level of written text: thus, for an input string at the level of underlying morphological elements, there will often be a unique output string or limited variations in output. For instance, *flatter* has two representations at the underlying level, but one spelling.

It is possible to do morphological analysis with bimachines, but one has to carry out linguistic description for morphological generation, and automatically derive morphological analysis from it. The method consists in compiling each bimachine (or commutative product of bimachines) into a transducer, and swapping input and output in the transducer. During the compilation of a bimachine into a transducer, the set of states of the transducer is constructed as the Cartesian product of the sets of states of the two automata.

3.1.9. Phonetic variations

Morphological analysis and generation of written text have an equivalent for speech: analysis and generation of phonetic forms. Phonetic forms are represented by strings of phonetic symbols. They describe how words are pronounced, taking into account contextual variants and free variants. An example of contextual phonetic variation in British English is the pronunciation of *more*, with r in *more ice* and without in *more tea*. Free variation is exemplified by *can* which can be either stressed or reduced in *He can see*. The input of analysis is thus a phonetic representation of speech. The output is some underlying representation of pronunciation, which is either conventional spelling, or a specific representation if additional information is needed, such as grammatical information.

The analysis of phonetic forms is useful for speech recognition. Their generation is useful for speech synthesis. A combination of both is a method for spelling correction: generate the pronunciation(s) of a misspelled word, then analyze the phonetic forms obtained.

A difference between phonetic processing and morphological processing is that a text can usually be pronounced in many ways, whereas spelling is much more standardized. In other aspects, the analysis and generation of phonetic forms is similar to morphological analysis and generation. The computational notions and tools involved are essentially the same.

The complexity of the task depends on the writing systems of languages. When all information needed to deduce phonetic strings, including information about phonetic variants, is encoded in spelling, then phonetic forms can be derived from written text without any recognition of the vocabulary. This is approximately the case of Spanish. Most Spanish words can be converted to phonetic strings by transducers, two-level rules or bimachines that do not comprise lexical information. Fig. 3.23 converts the letter c into the phonetic symbol θ before the vowels e and i.

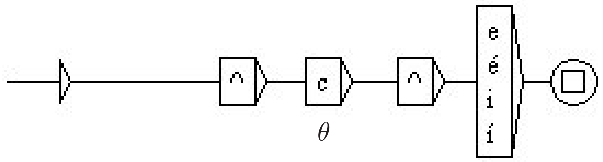


Figure 3.23. A phonetic conversion rule in Spanish.

In most of other languages, spelling is ambiguous: the pronunciation of a sequence of letters depends on the word in which it occurs in an unpredictable way. For example, *ea* between consonants is pronounced differently in *bead, head, beatific, creation, react*; in *read*, the pronunciation depends on the grammatical tense of the verb; in *lead*, it depends on the part of speech of the word: noun or verb. Due to such dependencies, which are most frequent in English and in French, phonetic forms cannot be generated from written texts accurately without vocabulary recognition. In other words, phonetic conversion requires a dictionary, which can be implemented in the form of a transducer and adapted for quick lookup into a generalized sequential transducer like that of Fig. 3.10.

However, even in languages with a disorderly writing system like English or French, the construction of such a dictionary can be partially automated. Transducers, two-level rules or bimachines can be used to produce tentative phonetic forms which have to be reviewed and validated or corrected by linguists.

A transducer that recognizes the vocabulary of a language is larger than a transducer that does not. They also differ in the way they delete word boundaries. In many languages, words are delimited in written text; they are not in phonetic strings, because speech is continuous and there is no audible evidence that a word ends and the next begins. In a transducer that recognizes the vocabulary, edges that delete word boundaries, e.g. edges labelled $(\# : \varepsilon)$, can be associated with ends of words. When the transducer is reversed by swapping input and output, the resulting transducer not only converts phonetics into spelling but also delimits words. The same cannot be done in a transducer that does not recognize vocabulary: since certain edge(s) erase word boundaries independently of context, the reversed transducer will generate optional word boundaries everywhere.

Phonetic strings are usually very ambiguous, and the result of their analysis consists of several hypotheses with different word delimitation, as in Fig. 3.24.

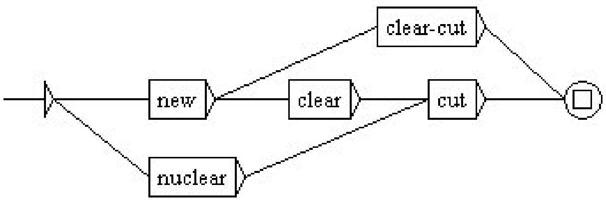


Figure 3.24. Acyclic automaton of the analyses of a phonetic form.

The result of the analysis of ambiguous input is naturally represented in an acyclic automaton like that of Fig. 3.24. We will call it an automaton of analyses, because it represents a set of mutually exclusive analyses. In language engineering, most specialists call such an automaton a "lattice"[5]. The output of a purely acoustic-to-phonetic phase of speech recognition is also an automaton of analyses: a segment of speech signal, i.e. the equivalent of a vowel or a consonant in acoustic signal, cannot always be definitely identified as a single phone (phonetic segment).

[5]This term has a precise mathematical meaning: an ordered set where each pair has a greatest lower bound and a least upper bound. As a matter of fact, in an acyclic graph, edges induce an ordering among the set of states. But the ordered set of states of an acyclic graph is not necessarily a lattice in the mathematical sense. In the acyclic automaton of Fig. 3.24, for instance, *cut* has no greatest lower bound and *new* has no least upper bound. Consequently we will avoid using the term "lattice" for denoting automata of analyses.

3.1.10. Weighted automata

The notions of automata and transducers exemplified in the preceding sections can be extended to weighted automata and transducers. In a weighted automaton, each transition has a weight which is an element of a semiring K; the set of terminal states is replaced with a terminal weight function from the set of states to K. The weight of a path is the product of the weights of its transitions. A Markov chain is a particular case of a weighted automaton.

In such models, weights approximate probabilities of occurrence of symbols in certain contexts, and the semiring is often $\mathbb{R}^+$. For example, in an automaton of analyses which contains phones recognized in a speech signal, weights can be assigned to each transition in order to represent the plausibility of the phone given the acoustic signal. The weighted automaton is exploited by selecting the path that maximizes the product of the weights.

Another example can be derived from Fig. 3.11: the plausibility of occurrence of a morphological element after a given left context could be added to this figure by assigning weights to boxes. The only known method of setting the value of these weights is based on statistics about occurrences of symbols or sequences in a sample of texts, a learning corpus.

Weighted automata are also used to compensate for the lack of accurate linguistic data. Weights are assigned to transitions in function of observable hints as to the occurrence of specific linguistic elements. During the analysis of a text, the weights are used to recognize those elements. For example, an initial uppercase letter is a hint of a proper name; the word ending *-ly* is a hint of an adverb like *shyly*. Weights are derived from statistics computed in a learning corpus. Results are inferior to those obtained with word lists of sufficient lexical coverage, e.g. lists of proper names or of adverbs: for instance, *bodily* ends in *-ly* but is usually an adjective. Word lists tend to be more and more used, but the two approaches are complementary, and the weighted-automaton method can make systems more robust when sufficiently extensive word lists are not available.

3.2. From words to sentences

3.2.1. Engineering approaches

The simplest model of the meaning of a text is the "word bag" model. Each word in the text represents an element of meaning, and the meaning of the text is represented by the set of the words that occur at least once in the text. The number of occurrences is usually attached to each word. The "word bag" model is used to perform tasks like content-based classification and indexation.

In order to implement the same tasks in a more elaborate way, or to implement other tasks, the sequential order of words must be taken into account. Translation is an example of an operation for which word order is obviously relevant: in many target languages, *The fly flies* and *The flies fly* should be translated differently. A model of text for which not only the value of words, but also their order, is relevant can be called a syntactic model. The formal and algorithmic tools involved in such a model depend entirely on the form of the linguistic data required. The most rational approach consists in constructing and using data similar to those mentioned in sections 3.1.4 to 3.1.9, but specifying ordered combinations of words. These data take the form of manually constructed lists or automata; some of them are automatically compiled into forms more adapted to computational operations. This approach is a long-term one. The stage of manual construction of linguistic data implies even more skill and effort than in the examples of section 3.1 (From letters to words), basically because there are many more words than letters. In addition, engineers feel uneasy with such data, that are largely outside their domain of competence; linguists feel uneasy with the necessary formal encoding; and little of the task can be automated. A consequence of this situation is a lack of linguistic resources that has been widely recognized, since 1990, as a major bottleneck in the development of language processing.

In order to avoid such work, alternative engineering techniques have been implemented and have had a dramatic development in recent years. The commonest of these techniques rely on weighted automata. (They are the most popular techniques based on weighted automata in language processing.) Weighted automata can be used to approximate various aspects of the grammar and syntax of languages: they can, for instance, guess at the part of speech of a word if the parts of speech of neighboring words are known. Weights are automatically derived from statistics about occurrences of symbols or sequences in a sample of texts, the learning corpus. The idea is similar to that with adverbs in *-ly* in section 3.1.10, but works even less well, for the same reason: there are more words than letters; there is a higher degree of complexity. As a matter of fact, in complex applications like translation and continuous speech recognition, results are still disappointing. Algorithms are well-known, but weights must be learnt for all words, and the only way of obtaining weights producing satisfactory results implies

- numerous occurrences of each word; therefore very large learning corpora (cf. section 3.1.1 about Zipf's law),
- statistics about sufficiently large contexts,
- sufficiently fine-grained tag sets.

The first constraint correctly predicts that if the learning corpus is too small, results are inadequate. When the size of the learning corpus increases, performances usually reach a maximum which is the best possible approximation in this framework. The last two constraints would lead to an explosion of the size of weighted automata and computational complexity. In practice, implementations of this method require considerable simplification of fundamental objects of the model: there is no serious attempt at processing compound words or ambiguity; the size of contexts is limited to two words to the left, and the size of tag sets to a few dozen tags, which is less than the tags et of Fig. 3.3. Finally, taking into consideration the third constraint would increase the cost of the manual tagging of the learning corpus, or require resorting to automatic tagging, with a corresponding output of inferior quality.

Resorting to such statistical approximations of grammar, syntax and the lexicon of languages is natural in so far as sufficiently accurate and comprehensive data seem out of reach. However, this is a short-term approach: it does not contribute to the enhancement of knowledge in these areas, and the technologies required for gathering exploitable and maintainable linguistic data have little in common with example-based learning. We can draw a parallel with meteorology: future weather depends on future physical data, or on physical data all around the world, including in marine areas where they are not measured with sufficient accuracy and frequency. Thus, weather is forecast on the basis of statistics about examples of past observations. However, designing weather forecast programs does not contribute to the advance of thermodynamics.

We will now turn to the linguistic approach. In order to relate formal notions with applications, we will refer primarily to translation, which is not a successfully automated operation yet, but which involves many of the basic operations in language processing.

3.2.2. Pattern definition and matching

Defining and matching patterns are two of these basic operations. In order to be able to translate a technical term like *microwave oven*, we must have a description of it, a method to locate occurrences in texts, and a link to a translation. The methods of description and location of such linguistic forms must take into account the existence of variants like the plural, *microwave ovens*, and possibly abbreviations like *MWO* if they are in use in relevant source texts. Thus, many linguistic forms are in fact sets of variants, and the actual form of all variants cannot always be computed from a canonical form. For example, the abbreviation *MWO* cannot be predicted from *microwave oven* by capitalizing initials, which would yield *MO*; the equivalence between *MWO* and the full form cannot be automatically inferred, even if the acronym occurs in a sample of source texts, because an explicit link between them, like *microwave oven (MWO)*, may be absent and, if present, would be ambiguous; etc. Thus the set of equivalent variants must often be manually constructed by linguists who are familiar with the field – a category of population which is often hard to find.

We can associate in a natural way *microwave oven* and its variants in the finite automaton of Fig. 3.25. When several lines are included in the same state, like *oven* and *ovens* here, they label parallel paths.

This type of automaton is more usual when there are more variants than with *microwave oven*. It is also used when the forms described are not equivalent, but constitute a small system which follows specific rules instead of general grammar rules of the language (Fig. 3.26). Such a system is called a local grammar.

In very restricted domains, the vocabulary and the syntactic constructions used in actual texts can be so stereotyped that all variability can be described in this form. This is the case of short stock exchange reports, weather forecast reports, sport scores etc. Local grammars can be used for translation, but this implies linking two monolingual local grammars together, one for the source language and another for the target language. Individual phrases of a grammar

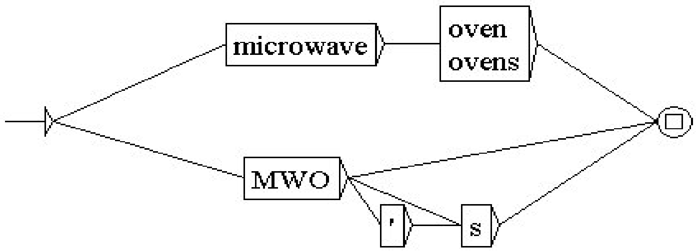


Figure 3.25. Definition of a simple linguistic pattern.

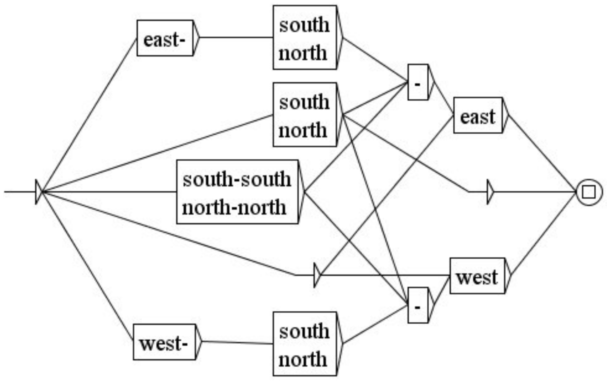


Figure 3.26. A local grammar.

must be specifically linked with phrases of the other, because they are not equivalent.

Finite automata defining linguistic patterns can be used to locate occurrences of the patterns in texts. When automata are as small as in the preceding instances, simple algorithms are sufficient: automata are compiled into the more traditional format with labelled edges and numbered states; they are determinized; they are matched against each point of the text.

A local grammar can be a representation of a subject of interest for a user in a text, for example one or several particular types of microwave ovens. In such a case, the local grammar can be used for text filtering, indexing and classification. Weights can be assigned to transitions in order to indicate the relevancy of paths with respect to the user's interest.

Comprehensive descriptions accounting for general language can reach impressive sizes. A complete grammar of dates, including informal dates, e.g. *before Christmas*, recognizes thousands of sequences. To be readable, such a description is necessarily organized into several automata. ¿From the formal point of view, the principle of such an organization is simple: a general finite automaton invokes sub-automata by special labels. Sub-automata, in turn, can equally invoke other sub-automata. Recursiveness may be allowed or not. In Fig. 3.27, the general automaton for numbers from 1 to 999 written in letters

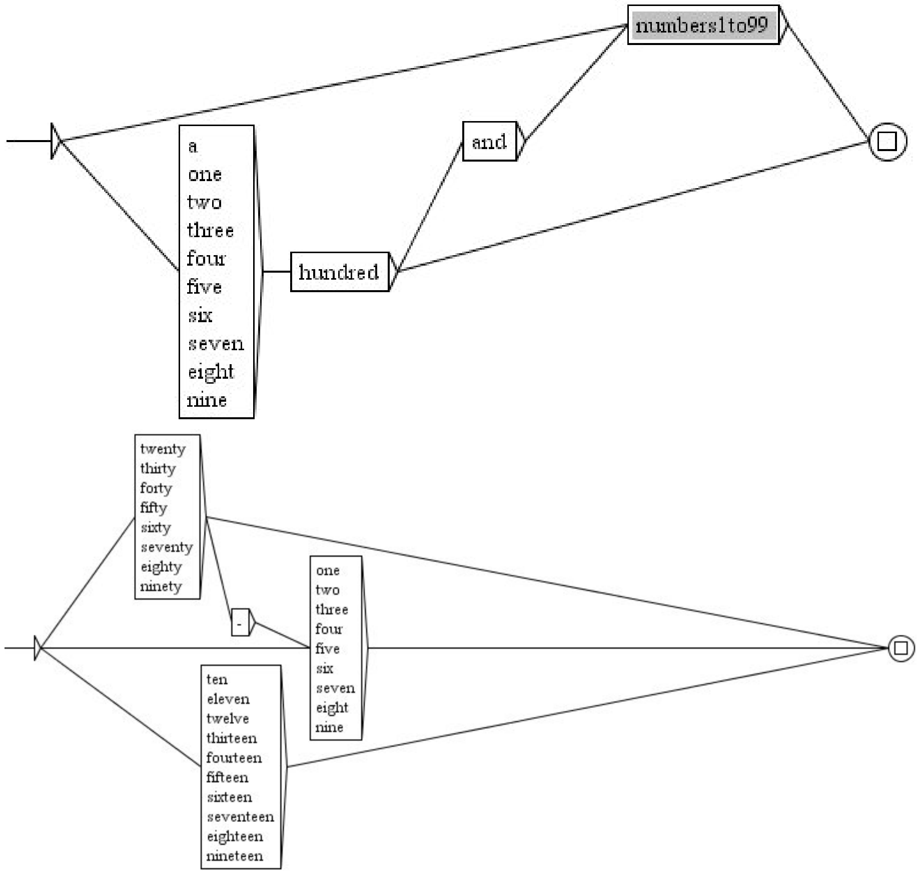


Figure 3.27. An automaton invokes another.

invokes the automaton for numbers from 1 to 99. The label for the second automaton is shown in grey. The use of labels for automata facilitates linguistic description for another reason: the same automaton can be invoked from several points and thus shared. Invoking an automaton via a label is thus equivalent to substituting it for the label. With patterns like terms, dates or numbers, invocations usually do not make up cycles: actual substitution is theoretically possible; it makes the set of automata equivalent to one finite automaton. However, with large grammars, actual substitution can lead to an explosion in size. For example, M. Gross's grammar of dates in French, which is organized into about 100 automata, becomes a 50-Mb automaton if sub-automata are systematically substituted. In the case of large grammars, the algorithms for locating occurrences in texts efficiently are therefore different: sub-automata are kept distinct and the matching algorithm is nondeterministic.

If cycles of invocations are allowed, the language recognized by the set of automata can be defined by reference to an equivalent context-free grammar (cf. section 1.6). The labels invoking sub-automata are the counterparts of variables, including the label of the general automaton which corresponds to the axiom of the grammar. Each of the automata is translated into a finite number of productions of the grammar. Such a set of automata is called a "recursive transition network" (RTN).

3.2.3. Parsing

If we consider more and more complex local grammars, we reach a point where the identification of a linguistic form depends on the identification of free constituents. Free constituents are syntactic constructs, like sentences or noun phrases, which involve open categories, like verbs or nouns, in their content. For example, recognizing the phrase *take into account* may imply identifying:

- its subject, which cannot be any noun, e.g. not *air*, and
- its free complement, which can occur before or after *into account*.

Both are free constituents. The subject is a noun phrase, which comprises at least an open category, a noun. The free complement can be a noun phrase or a sentential clause: *Max took into account that Mary was early.* The identification of these free and frozen constituents is required for complex applications like translation.

Several features of RTNs make them adequate for the formal description of such phrases.

- Free constituents can be represented by labels invoking other parts of the grammar. In the example of *take into account*, these labels will represent types of noun phrases, of sentences and of sentential clauses. Obviously, the labels are reusable from other points of the grammar, because other phrases or verbs will accept the same types of subjects or of complements.
- Small lexical variations and alternative constructions are described in parallel paths of the automata, as in Fig. 3.28.
- Recursiveness can be used for embeddings between syntactic constructs. In the example of Fig. 3.28, the phrase and the free constituents around it make up a sentence; the label S included in the automaton represents sentences. Thus, the rule is recursive.

A large variety of syntactic constructions in natural languages can be expressed in that way. A complete description of *take into account*, for example, should include passive, interrogative forms etc., and would be much larger than

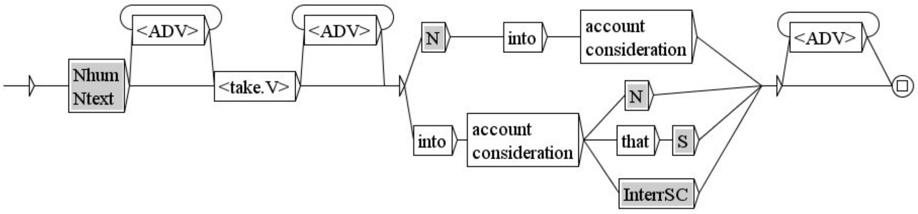


Figure 3.28. A sample of a grammar of *take into account*.

this figure. In addition, the number of grammatical constructions in a language is in some way multiplied by the size of the lexicon, since different words do not enter into the same grammatical constructions. However, the construction of large grammars for thousands of phrases and verbs can be partially automated. General grammars are manually constructed in the form of parameterized RTNs, then they are adapted to specific lexical items like *take into account* by setting the values of the parameters. These values are encoded for each lexical item in tables of syntactic properties. A large proportion of the parameters must be at the level of specific lexical items, and not of classes of items (e.g. transitive verbs), because syntactic properties are incredibly dependent on actual lexical items.

Here are two examples of open problems in the construction of grammars[6]: selectional constraints between predicates (i.e. verbs, nouns and adjectives) and their arguments (i.e. subject and essential complements):

*(Max + *The air) took into account that Mary was early*

and selectional constraints between predicates and adverbs:

*Max took the delay into account (last time + *by plane)*

Present grammars either overgenerate or undergenerate when such constraints come into play.

Even so, the construction of grammars of natural languages in the form of RTNs now appears to be within reach.

This situation provides partial answers for a classical controversy about the most popular two formal models of syntax: finite automata and context-free grammars. The issue of the adequacy of these two models dates back to the time of their actual definition and is still going on. Infrequent constructions have been used to argue that both were inadequate, but they can be conveniently dealt with as exceptions. ¿From 1960 to 1990, the folklore of the domain held that it was reasonable practice to use context-free grammars, and a heresy to use automata. Since then, investigation results suggested that the RTN model, which is equivalent to grammars but relies heavily on the automaton form, is

[6]In the next two examples, the star * marks that a sequence is not acceptable as a sentence.

convenient for the manual description of syntax as well as for automatic parsing. It is an open question as to whether the non-recursive counterpart of RTNs, which is equivalent to finite automata, will be better. Recursiveness can surely be eliminated from RTNs through an automatic compilation process, by substituting cycles for terminal embeddings and by limiting central embeddings to a fixed maximal depth. But even without recursiveness, RTN-based parsing is not necessarily more similar to automaton-based parsing than context-free parsing... In any case, the issue now appears less theoretical than computational.

3.2.4. Lexical ambiguity reduction

We mentioned lexical tagging in section 3.1.4. This operation consists of assigning tags to words. Word tags record linguistic information. Lexical tagging is not an application in itself, since word tags contain encoded information not directly exploitable by users. However, lexical tagging is required for enhancing the results of nearly all operations on texts: translation, spelling correction, location of index terms etc. Section 3.1.4 shows how dictionary lookup contributes to lexical tagging, but many words should be assigned distinct tags in relation to context, like *record*, a noun or a verb. Such forms are said to be lexically ambiguous. Syntactic parsing often resolves all lexical ambiguity. Sentences like the following are rare:

The newspapers found out some record

This ambiguous sentence has two syntactic analyses: *some record* is a noun phrase or a sentential clause, and *record* is accordingly a noun or a verb.

Syntactic parsing is not a mature technique yet, and there is a need for procedures that can work without complete syntactic grammars of languages, even if they resolve less lexical ambiguity than syntactic parsing.

Such a procedure can be designed on the following basis. After dictionary lookup, a text can be represented as an acyclic automaton of analyses like that of Fig. 3.29. Syntactic constraints can be represented as an automaton over the

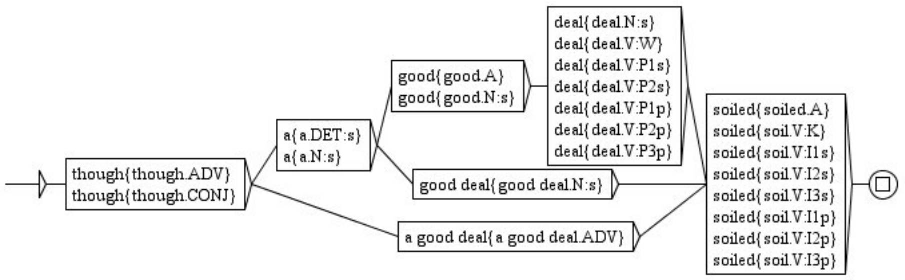


Figure 3.29. The automaton of analyses of *though a good deal soiled.*

same alphabet. Fig. 3.30 states that when the word *good* is a noun, it cannot follow the indefinite determiner *a*. The label @ stands for a default symbol:

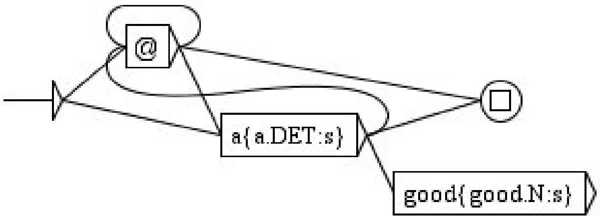


Figure 3.30. An automaton stating a syntactic constraint.

it matches the next input symbol if, at this point of the automaton, no other symbol matches. The intersection of the two automata is shown in Fig. 3.31; it

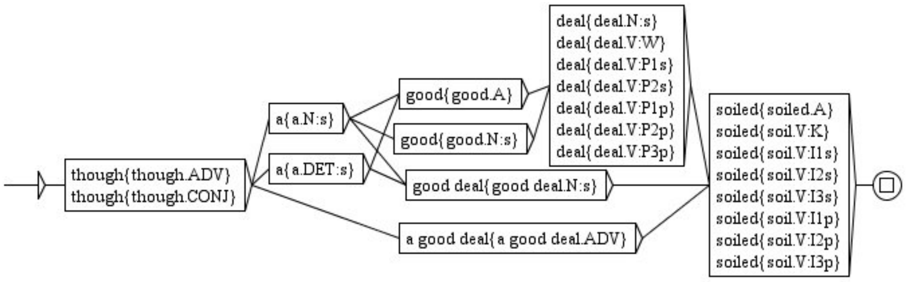


Figure 3.31. The intersection of the two automata.

represents those analyses of the text that obey the constraints. The intersection of two automata is an automaton that recognizes the intersection of the two languages recognized. It is constructed by a simple algorithm. Different syntactic constraints can be represented by different automata: since intersection is associative and commutative, the automata can be intersected in any order without changing the result. Thus, various syntactic constraints can be formalized independently and accumulated in order to reduce progressively more lexical ambiguity. However, this approach needs a convenient interface to allow linguists to express the constraints in the form of automata. Automata like that of Fig. 3.30 can be directly constructed only in very simple cases.

An alternative approach combines dictionary lookup and ambiguity resolution in another way. It considers that the relevant data are (i) the probability for a given word to occur with a given tag, and (ii) the probability of occurrence of a sequence of words (or tags). Such probabilities are estimated on the basis of statistics in a tagged corpus. The resulting values are inserted into a weighted automaton to make up a model of language. This technique has been applied to small tag sets, and the possibility of tagging compound words has not been seriously investigated.

Notes

The notion of formal model in linguistics emerged progressively. We will mention a few milestones on this path. During the first half of the twentieth century, Saussure stated clearly that language is a system and that form/meaning associations are arbitrary. This was a first step towards the separation between syntax and semantics. The translation of this idea into practice owes much to the study of native American languages by Sapir 1921. During the second half of the century, Harris incorporated the information aspect into the study of the forms of language. In particular, he introduced the notion of transformation (Harris 1952, Harris 1970). Gross 1975, Gross 1979 originated the construction of tables of syntactic properties. The parameterized graphs of section 3.2.3 are used in Senellart 1998 and Paumier 2001.

The theory of formal languages developed in parallel (Schützenberger and Chomsky 1963; Gross and Lentin 1967). Discussions arose during the same period of time about the adequacy of formal models for representing the behavior of speakers (Miller and Chomsky 1963) or the syntax of natural languages. Chomsky 1956, Chomsky 1957 mathematically "proved" that neither finite automata nor context-free languages were adequate for syntax, but he used infrequent constructions that can be conveniently dealt with as exceptions (Gross 1995). Gross gave an impulse to the actual production of extensive descriptions of lexicon and syntax with finite automata.

The observations that led to the statement of Zipf's law (Zipf 1935) were not restricted to language. The results exposed in section 3.1.3 about Zipf's law applied to written texts are based on Senellart 1999.

Johnson 1972 investigated various ways of combining formal rules and established whether the result of combination can be represented as a finite automaton. The notion of sequential transducer originates from Schützenberger 1977. Two algorithms of minimization of sequential transducers are known (Breslauer 1998; Béal and Carton 2001); the second one is based on successive contributions by Choffrut 1979, Reutenauer 1990 and Mohri 1994 (see also Chapter 1). The definition of p-sequential transducers was proposed by Mohri 1994. The algorithm of construction of generalized sequential transducers is adapted from Roche 1997.

The representation of finite automata as graphs with labels attached to states was introduced into language processing by Gross 1989 and Silberztein 1994 (`http://acl.ldc.upenn.edu/C/C94/C94-1095.pdf`). The Unitex system (`http://www-igm.univ-mlv.fr/~unitex`), implemented by Sébastien Paumier at the University of Marne-la-Vallée, is an open-source environment for language processing with automata and dictionaries.

The use of the intersection of finite transducers for specifying and implementing morphological analysis and generation, and for lexical ambiguity resolution, was first suggested by Koskenniemi 1983. Bimachines were introduced by Schützenberger 1961. The adaptation of bimachines to morphology and phonetics comes from Laporte 1997.

Weighted automata and transducers are defined by Paz 1971 and Eilenberg 1974. The FSM library (Mohri, Pereira, and Riley 2000) offers consistent tools related to weighted automata.

Algorithms for deriving weights from statistics about occurrences of symbols or sequences in a learning corpus are available in handbooks, e.g. Jurafsky and Martin 2000.