2

Corpus Processing Tools

2.1 Corpora

A corpus, plural corpora, is a collection of texts or speech stored in an electronic machine-readable format. A few years ago, large electronic corpora of more than a million of words were rare, expensive, or simply not available. At present, huge quantities of texts are accessible in many languages of the world. They can easily be collected from a variety of sources, most notably the Web, where corpora of hundreds of millions of words are within the reach of most computational linguists.

2.1.1 Types of Corpora

Some corpora focus on specific genres, law, science, novels, news broadcasts, transcriptions of telephone calls, or conversations. Others try to gather a wider variety of running texts. Texts collected from a unique source, say from scientific magazines, will probably be slanted toward some specific words that do not appear in everyday life. Table 2.1 compares the most frequent words in the book of Genesis and in a collection of contemporary running texts. It gives an example of such a discrepancy. The choice of documents to include in a corpus must then be varied to survey comprehensively and accurately a language usage. This process is referred to as balancing a corpus.

Balancing a corpus is a difficult and costly task. It requires collecting data from a wide range of sources: fiction, newspapers, technical, and popular literature. Balanced corpora extend to spoken data. The Linguistic Data Consortium from the University of Pennsylvania and The European Language Resources Association (ELRA), among other organizations, distribute written and spoken corpus collections. They feature samples of magazines, laws, parallel texts in English, French, German, Spanish, Chinese, telephone calls, radio broadcasts, etc.

In addition to raw texts, some corpora are annotated. Each of their words is labeled with a linguistic tag such as a part of speech or a semantic category. The annotation is done either manually or semiautomatically. Spoken corpora contain the

Table 2.1. List of the most frequent words in present texts and in the book of Genesis. After Crystal (1997).

	English	French	German
Most frequent words in a collection of contemporary running texts	*the*	*de*	*der*
	of	*le* (article)	*die*
	to	*la* (article)	*und*
	in	*et*	*in*
	and	*les*	*des*
Most frequent words in Genesis	*and*	*et*	*und*
	the	*de*	*die*
	of	*la*	*der*
	his	*à*	*da*
	he	*il*	*er*

transcription of spoken conversations. This transcription may be aligned with the speech signal and sometimes includes prosodic annotation: pause, stress, etc. Annotation tags, paragraph and sentence boundaries, parts of speech, syntactic or semantic categories follow a variety of standards, which are called markup languages.

Among annotated corpora, treebanks deserve a specific mention. They are collections of parse trees or more generally syntactic structures of sentences. The production of a treebank generally requires a team of linguists to parenthesize the constituents of a corpus or to arrange them in a structure. Annotated corpora require a fair amount of handwork and are therefore more expensive than raw texts. Treebanks involve even more clerical work and are relatively rare. The Penn Treebank (Marcus et al. 1993) from the University of Pennsylvania is a widely cited example for English.

A last word on annotated corpora: in tests, we will benchmark automatic methods against manual annotation, which is often called the Gold Standard. We will assume the hand annotation perfect, although this is not true in practice. Some errors slip into hand-annotated corpora, even in those of the best quality, and the annotators may not agree between them. The scope of agreement varies depending on the annotation task. The inter-annotator agreement is high for parts of speech. It is lower when annotating the sense of a word.

2.1.2 Corpora and Lexicon Building

Lexicons and dictionaries are intended to give word lists, to provide a reader with word senses and meanings, and to outline their usage. Dictionaries' main purpose is related to lexical semantics. Lexicography is the science of building lexicons and writing dictionaries. It uses electronic corpora extensively.

The basic data of a dictionary is a word list. Such lists can be drawn manually or automatically from corpora. Then, lexicographers write the word definitions and choose citations illustrating the words. Since most of the time current meanings are

obvious to the reader, meticulous lexicographers tended to collect examples – citations – reflecting a rare usage. Computerized corpora can help lexicographers avoid this pitfall by extracting all the citations that exemplify a word. An experienced lexicographer will then select the most representative examples that reflect the language with more relevance. S/he will prefer and describe more frequent usage and possibly set aside others.

Finding a citation involves sampling a fragment of text surrounding a given word. In addition, the context of a word can be more precisely measured by finding recurrent pairs of words, or most frequent neighbors. The first process results in concordance tables, and the second one in collocations.

Concordance tables were first produced for antiquity and religious studies. Hugues de Saint Cher is known to have compiled the first Bible concordance in the thirteenth century. Concordances consist of text excerpts centered on a specific word and surrounded by a limited number of words before and after it (Table 2.2). Other more elaborate concordances take word morphology into account or group words together into semantic themes. Sœur Jeanne d'Arc (1970) produced an example of such a concordance for Bible studies.

Table 2.2. Concordance of *miracle* in the Gospel of John.

Language	Concordances
English	s beginning of miracles did Je n they saw the miracles which n can do these miracles that t ain the second miracle that Je e they saw his miracles which
French	le premier des miracles que fi i dirent: Quel miracle nous mo om, voyant les miracles qu'il peut faire ces miracles que tu s ne voyez des miracles et des
German	ist das erste Zeichen, das Je du uns für ein Zeichen, daß du en, da sie die Zeichen sahen, emand kann die Zeichen tun, di Wenn ihr nicht Zeichen und Wun

Concordancing is a powerful tool to study usage patterns and to write definitions. It also provides evidences on certain preferences between verbs and prepositions, adjectives and nouns, recurring expressions, or common syntactic forms. These couples are referred to as **collocations**. Church and Mercer (1993) cite a striking example of idiosyncratic collocations of *strong* and *powerful*. While *strong* and *powerful* have similar definitions, they occur in different contexts, as shown in Table 2.3.

Table 2.4 shows additional collocations of *strong* and *powerful*. These word preferences cannot be explained using rational definitions, but can be observed in cor-

Table 2.3. Comparing *strong* and *powerful*.

	English	**French**	**German**
You say	*Strong tea*	*Thé fort*	*Kräftiger Tee*
	Powerful computer	*Ordinateur puissant*	*Starker Computer*
You don't say	*Strong computer*	*Thé puissant*	*Starker Tee*
	Powerful tea	*Ordinateur fort*	*Kräftiger Computer*

pora. A variety of statistical tests can measure the strength of pairs, and we can extract them automatically from a corpus.

Table 2.4. Word preferences of *strong* and *powerful* collected from the Associated Press corpus. Numbers in columns indicate the number of collocation occurrences with word w. After Church and Mercer (1993).

Preference for *strong* over *powerful*			**Preference for *powerful* over *strong***		
strong w	*powerful* w	w	*strong* w	*powerful* w	w
161	0	*showing*	1	32	*than*
175	2	*support*	1	32	*figure*
106	0	*defense*	3	31	*minority*
...					

2.1.3 Corpora as Knowledge Sources for the Linguist

In the beginning of the 1990s, computer-based corpus analysis completely renewed empirical methods in linguistics. It helped design and implement many of the techniques presented in this book. As we saw with dictionaries, corpus analysis helps lexicographers acquire lexical knowledge and describe language usage. More generally, corpora enable us to experiment with tools and to confront theories on real data. For most language analysis programs, collecting relevant corpora of texts has then become a necessary step to define specifications and measure performances. Let us take the examples of part-of-speech taggers, parsers, and dialogue systems.

Annotated corpora are essential tools to develop part-of-speech taggers or parsers. A first purpose is to measure the tagging or parsing performance. The tagger or parser is run on texts and their result is compared to hand annotation, which serves as a reference. A linguist or an engineer can then determine the accuracy, the robustness of an algorithm or a parsing model and see how well it scales up by applying it to a variety of texts.

A second purpose of annotated corpora is to be a knowledge source to refine tagging techniques and improve grammars. While developing a grammar, a linguist can see if changing a rule improves or deteriorates results. The tool tuning is then done manually. Using statistical techniques, annotated corpora also enable researchers to

acquire grammar rules or language models automatically or semiautomatically to tag or parse a text. We will see this in Chap. 7.

A dialogue corpus between a user and a machine is also critical to develop an interactive spoken system. The corpus is usually collected through fake dialogues between a real user and a person simulating the machine answers. Repeating such experiments with a reasonable number of users enables us to acquire a text set covering what the machine can expect from potential users. It is then easier to determine the vocabulary of an application, to have a precise idea of word frequencies, and to know the average length of sentences. In addition, the dialogue corpus enables the analyst to understand what the user expects from the machine, that is, how s/he interacts with it.

2.2 Finite-State Automata

2.2.1 A Description

The most frequent operation we do with corpora consists in searching words or phrases. To be convenient, search must extend beyond fixed strings. We may want to search a word or its plural form, uppercase or lowercase letters, expressions containing numbers, etc. This is made possible using finite-state automata (FSA) that we introduce now. FSA are flexible tools to process texts and one of the most adequate to search strings.

FSA theory was designed in the beginning of computer science as a model of abstract computing machines. It forms a well-defined formalism that has been tested and used by generations of programmers. FSA stem from a simple idea. These are devices that accept – recognize – or reject an input stream of characters. FSA are very efficient in terms of speed and memory occupation and are easy to implement in Prolog. In addition to text searching, they have many other applications: morphological parsing, part-of-speech annotation, and speech processing.

Figure 2.1 shows a three-state automaton numbered from 0 to 2, where state q_0 is called the start state and q_2 the final state. An automaton has a single start state and any number of final states, indicated by double circles. Arcs between states designate the possible transitions. Each arc is annotated by a label, which means that the transition accepts or generates the corresponding character.

An automaton accepts an input string in the following way: it starts in the initial state, follows a transition where the arc character matches the first character of the string, consumes the corresponding string character, and reaches the destination state. It makes then a second transition with the second string character and continues in this way until it ends up in one of the final states and there is no character left. The automaton in Fig. 2.1 accepts or generates strings such as: ac, abc, $abbc$, $abbbc$, $abbbbbbbbbbbc$, etc. If the automaton fails to reach a final state, either because it has no more characters in the input string or because it is trapped in a nonfinal state, it rejects the string.

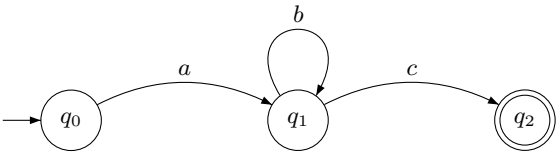

Fig. 2.1. A finite-state automaton.

As an example, let us see how the automaton accepts string $abbc$ and rejects $abbcb$. The input $abbc$ is presented to the start state q_0. The first character of the string matches that of the outgoing arc. The automaton consumes character a and moves to state q_1. The remaining string is bbc. Then, the automaton loops twice on state q_1 and consumes bb. The resulting string is character c. Finally, the automaton consumes c and reaches state q_2, which is the final state. On the contrary, the automaton does not accept string $abbcb$. It moves to states q_0, q_1, and q_2, and consumes $abbc$. The remaining string is letter b. Since there is no outgoing arc with a matching symbol, the automaton is stuck in state q_2 and rejects the string.

Automata may contain ε-transitions from one state to another. In this case, the automaton makes a transition without consuming any character of the input string. The automaton in Fig. 2.2 accepts strings a, ab, abb, etc. as well as ac, abc, $abbc$, etc.

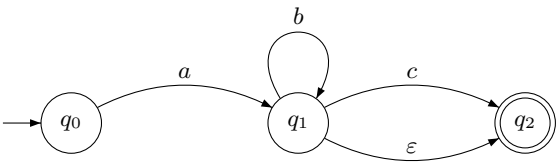

Fig. 2.2. A finite-state automaton with an ε-transition.

2.2.2 Mathematical Definition of Finite-State Automata

FSA have a formal definition. An FSA consists of five components $(Q, \Sigma, q_0, F, \delta)$, where:

1. Q is a finite set of states.
2. Σ is a finite set of symbols or characters: the input alphabet.
3. q_0 is the start state, $q_0 \in Q$.
4. F is the set of final states, $F \subseteq Q$.
5. δ is the transition function $Q \times \Sigma \rightarrow Q$, where $\delta(q, i)$ returns the state where the automaton moves when it is in state q and consumes the input symbol i.

The quintuple defining the automaton in Fig. 2.1 is $Q = \{q_0, q_1, q_2\}$, $\Sigma = \{a, b, c\}$, $F = \{q_2\}$, and $\delta = \{\delta(q_0, a) = q_1, \delta(q_1, b) = q_1, \delta(q_1, c) = q_2\}$. The state-transition table in Table 2.5 is an alternate representation of the δ function.

Table 2.5. A state-transition table where $\emptyset$ denotes nonexisting or impossible transitions.

State\Input	**a**	**b**	**c**
q_0	q_1	$\emptyset$	$\emptyset$
q_1	$\emptyset$	q_1	q_2
q_2	$\emptyset$	$\emptyset$	$\emptyset$

2.2.3 Finite-State Automata in Prolog

A finite-state automaton has a straightforward implementation in Prolog. It is merely the transcription of the quintuplet definition. The following code describes the transitions, the start, and the final states of the automaton in Fig. 2.1:

```
% The start state
start(q0).

% The final states
final(q2).

% The transitions
% transition(SourceState, Symbol, DestinationState)
transition(q0, a, q1).
transition(q1, b, q1).
transition(q1, c, q2).
```

The predicate `accept/1` selects the start state and runs the automaton using `accept/2`. The predicate `accept/2` is recursive. It succeeds when it reaches a final state, or consumes a symbol of the input string and makes a transition otherwise.

```
accept(Symbols) :-
  start(StartState),
  accept(Symbols, StartState).

% accept(+Symbols, +State)
accept([], State) :-
  final(State).
accept([Symbol | Symbols], State) :-
  transition(State, Symbol, NextState),
  accept(Symbols, NextState).
```

`accept/1` either accepts an input symbol string or fails:

```
?- accept([a, b, b, c]).
Yes

?- accept([a, b, b, c, b]).
No
```

The automaton in Fig. 2.2 contains ε-transitions. They are introduced in the database as facts:

```
epsilon(q1, q2).
```

To take them into account, the `accept/2` predicate should be modified so that there are two possible sorts of transitions. A first rule consumes a character and a second one, corresponding to an ε-transition, passes the string unchanged to the next state:

```
accept([], State) :-
  final(State).
accept([Symbol | Symbols], State) :-
  transition(State, Symbol, NextState),
  accept(Symbols, NextState).
accept(Symbols, State) :-
  epsilon(State, NextState),
  accept(Symbols, NextState).
```

2.2.4 Deterministic and Nondeterministic Automata

The automaton in Fig. 2.1 is said to be deterministic (DFSA) because given a state and an input, there is one single possible destination state. On the contrary, a nondeterministic automaton (NFSA) has states where it has a choice: the path is not determined in advance.

Figure 2.3 shows an example of an NFSA that accepts the strings ab, abb, $abbb$, $abbbb$, etc. Taking abb as input, the automaton reaches the state q_1 consuming the letter a. Then, it has a choice between two states. The automaton can either move to state q_2 or stay in state q_1. If it first moves to state q_2, there will be one character left and the automaton will fail. The right path is to loop onto q_1 and then to move to q_2. ε-transitions also cause automata to be nondeterministic as in Fig. 2.2 where any string that has reached state q_1 can also reach state q_2.

A possible strategy to deal with nondeterminism is to use backtracking. When an automaton has the choice between two or more states, it selects one of them and remembers the state where it made the decision: the choice point. If it subsequently fails, the automaton backtracks to the choice point and selects another state to go to. In our example in Fig. 2.3, if the automaton moves first to state q_2 with the string *bb*, it will end up in a state without outgoing transition. It will have to backtrack and select state q_1. Backtracking is precisely the strategy that Prolog uses automatically.

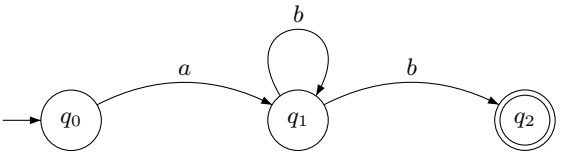

Fig. 2.3. A nondeterministic automaton.

2.2.5 Building a Deterministic Automata from a Nondeterministic One

Although surprising, any nondeterministic automaton can be converted into an equivalent deterministic automaton. We outline here an informal description of the determinization algorithm. See Hopcroft et al. (2001) for a complete description of this algorithm.

The algorithm starts from an NFSA $(Q_N, \Sigma, q_0, F_N, \delta_N)$ and builds an equivalent DFSA $(Q_D, \Sigma, \{q_0\}, F_D, \delta_D)$, where:

- Q_D is the set of all the possible state subsets of Q_N. It is called the power set. The set of states of the automaton in Fig. 2.3 is $Q_N = \{q_0, q_1, q_2\}$. The corresponding set of sets is $Q_D = \{\emptyset, \{q_0\}, \{q_1\}, \{q_2\}, \{q_0, q_1\}, \{q_0, q_2\}, \{q_1, q_2\}, \{q_0, q_1, q_2\}\}$. If Q_N has n states, Q_D will have 2^n states. In general, many of these states will be inaccessible and will be discarded.
- F_D is the set of sets that include at least one final state of F_N. In our example, $Q_D = \{\{q_2\}, \{q_0, q_2\}, \{q_1, q_2\}, \{q_0, q_1, q_2\}\}$.
- For each set $S \subset Q_N$ and for each input symbol a, $\delta_D(S, a) = \bigcup_{s \in S}^{\delta_N(s,a)}$. The state-transition table in Table 2.6 represents the automaton in Fig. 2.3. Table 2.7 represents the determinized version of it.

Table 2.6. The state-transition table of the nondeterministic automaton shown in Fig. 2.3.

State\Input	a	b
q_0	q_1	$\emptyset$
q_1	$\emptyset$	q_1, q_2
q_2	$\emptyset$	$\emptyset$

2.2.6 Searching a String with a Finite-State Automaton

Searching the occurrences of a string in a text corresponds to recognizing them with an automaton, where the string characters label the sequence of transitions. However, the automaton must skip chunks in the beginning, between the occurrences, and at

Table 2.7. The state-transition table of the determinized automaton in Fig. 2.3.

State\Input	**a**	**b**
$\emptyset$	$\emptyset$	$\emptyset$
$\{q_0\}$	$\{q_1\}$	$\emptyset$
$\{q_1\}$	$\emptyset$	$\{q_1, q_2\}$
$\{q_2\}$	$\emptyset$	$\emptyset$
$\{q_0, q_1\}$	$\{q_1\}$	$\{q_1, q_2\}$
$\{q_0, q_2\}$	$\{q_1\}$	$\emptyset$
$\{q_1, q_2\}$	$\emptyset$	$\{q_1, q_2\}$
$\{q_0, q_1, q_2\}$	$\{q_1\}$	$\{q_1, q_2\}$

the end of the text. The automaton consists then of a core accepting the searched string and of loops to process the remaining pieces. Consider again the automaton in Fig. 2.1 and modify it to search strings ac, abc, $abbc$, $abbbc$, etc., in a text. We add two loops: one in the beginning and the other to come back and start the search again (Fig. 2.4).

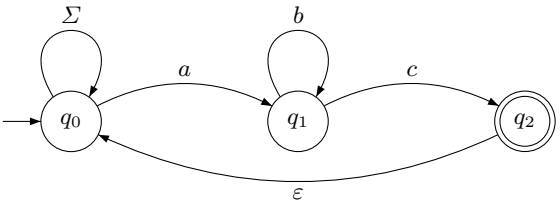

Fig. 2.4. Searching strings ac, abc, $abbc$, $abbbc$, etc.

In doing this, we have built an NFSA that it is preferable to convert into a DFSA. Hopcroft et al. (2001) describe the mathematical properties of such automata and an algorithm to automatically build an automaton for a given set of patterns to search. They notably report that resulting DFSA have exactly the same number of states as the corresponding NFSA. We present an informal solution to determine the transitions of the automaton in Fig. 2.4.

If the input text does not begin with an a, the automaton must consume the beginning characters and loop on the start state until it finds one. Figure 2.5 expresses this with an outgoing transition from state 0 to state 1 labeled with an a and a loop for the rest of the characters. $\Sigma - a$ denotes the finite set of symbols except a. From state 1, the automaton proceeds if the text continues with either a b or a c. If it is an a, the preceding a is not the beginning of the string, but there is still a chance because it can start again. It corresponds to the second loop on state 1. Otherwise, if the next character falls in the set $\Sigma - \{a, b, c\}$, the automaton goes back to state 0. The automaton successfully recognizes the string if it reaches state 2. Then it goes

back to state 0 and starts the search again, except if the next character is an a, for which it can go directly to state 1.

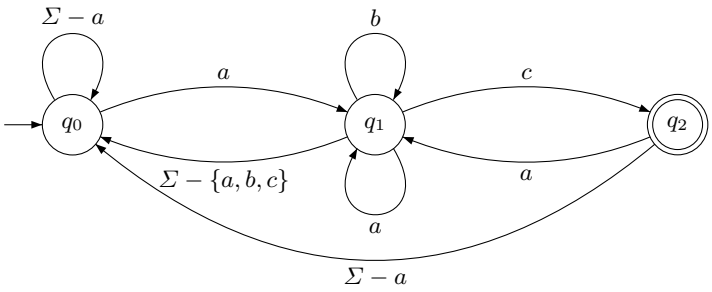

Fig. 2.5. An automaton to search strings ac, abc, $abbc$, $abbbc$, etc., in a text.

2.2.7 Operations on Finite-State Automata

FSA can be combined using a set of operations. The most useful are the union, the concatenation, and the closure.

The union or sum of two automata A and B accepts or generates all the strings of A and all the strings of B. It is denoted $A \cup B$. We obtain it by adding a new initial state that we link to the initial states of A and B (Fig. 2.6) using ε-transitions (Fig. 2.7).

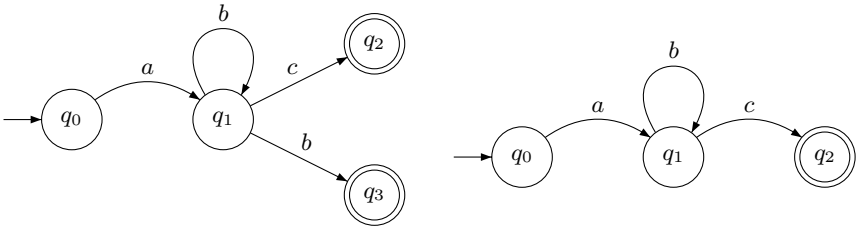

Fig. 2.6. Automata A (left) and B (right).

The concatenation or product of A and B accepts all the strings that are concatenations of two strings, the first one being accepted by A and the second one by B. It is denoted $A.B$. We obtain the resulting automaton by connecting all the final states of A to the initial state of B using ε-transitions (Fig. 2.8).

The iteration or Kleene closure of an automaton A accepts the concatenations of any number of its strings and the empty string. It is denoted A^*, where $A^* = \{\varepsilon\} \cup A \cup A.A \cup A.A.A \cup A.A.A.A \cup \ldots$. We obtain the resulting automaton by linking the final states of A to its initial state using ε-transitions and adding a new

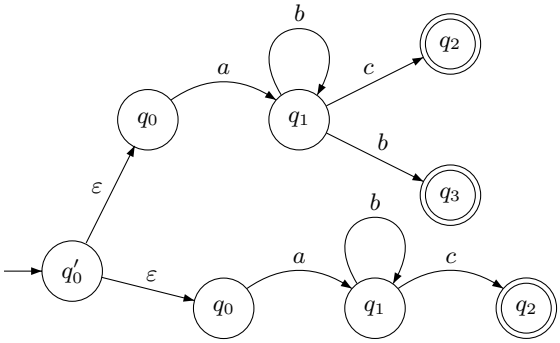

Fig. 2.7. The union of two automata: $A \cup B$.

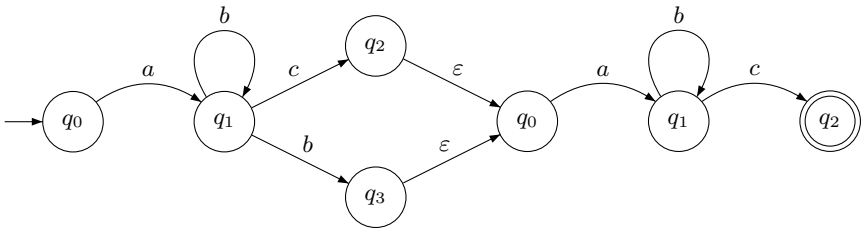

Fig. 2.8. The concatenation of two automata: $A.B$.

initial state, as shown in Fig. 2.9. The new initial state enables us to obtain the empty string.

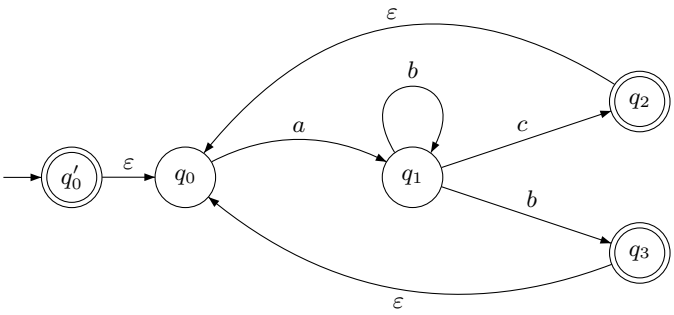

Fig. 2.9. The closure of A.

The notation Σ^* designates the infinite set of all possible strings generated from the alphabet Σ. Other significant operations are:

- The intersection of two automata $A \cap B$ that accepts all the strings accepted both by A and by B. If $A = (\Sigma, Q_1, q_1, F_1, \delta_1)$ and $B = (\Sigma, Q_2, q_2, F_2, \delta_2)$, the resulting automaton is obtained from the Cartesian product of states $(\Sigma, Q_1 \times Q_2, \langle q_1, q_2 \rangle, F_1 \times F_2, \delta_3)$ with the transition function $\delta_3(\langle s_1, s_2 \rangle, i) = \{\langle t_1, t_2 \rangle \mid t_1 \in \delta_1(s_1, i) \wedge t_2 \in \delta_2(s_2, i)\}$.
- The difference of two automata $A - B$ that accepts all the strings accepted by A but not by B.
- The complementation of the automaton A in Σ^* that accepts all the strings that are not accepted by A. It is denoted $\bar{A}$, where $\bar{A} = \Sigma^* - A$.
- The reversal of the automaton A that accepts all the reversed strings accepted by A.

Two automata are said to be equivalent when they accept or generate exactly the same set of strings. Useful equivalence transformations optimize computation speed or memory requirements. They include:

- ε-removal, which transforms an initial automaton into an equivalent one without ε-transitions
- determinization, which transforms a nondeterministic automaton into a deterministic one
- minimization, which determines among equivalent automata the one that has the smallest number of states

Optimization algorithms are out of the scope of this book. Hopcroft et al. (2001) as well as Roche and Schabes (1997) describe them in detail.

2.3 Regular Expressions

The automaton in Fig. 2.1 generates or accepts strings composed of one a, zero or more b's, and one c. We can represent this set of strings using a compact notation: `ab*c`, where the star symbol means any number of the preceding character. Such a notation is called a regular expression or regex. Regular expressions are very powerful devices to describe patterns to search in a text. Although their notation is different, regular expressions can always be implemented under the form of automata, and vice versa. However, regular expressions are generally easier to use.

Regular expressions are composed of literal characters, that is, ordinary text characters like `abc`, and of metacharacters like `*` that have a special meaning. The simplest form of regular expressions is a sequence of literal characters: letters, numbers, spaces, or punctuation signs. Regexes `regular` or `Prolog` match strings *regular* or *Prolog* contained in a text. Table 2.8 shows examples of pattern matching with literal characters. Regular expressions are case-sensitive and match the first instance of the string or all its instances in a text, depending on the regex language that is used.

There are currently a dozen major regular expression languages freely available. Their common ancestor is `grep`, which stands for global/regular expression/print.

Table 2.8. Examples of simple patterns and matching results.

Pattern	String
`regular`	"A section on regular expressions"
`Prolog`	"The Prolog language"
`the`	"The book of the life"

`grep` is a standard Unix tool that prints out all the lines of a file that contain a given pattern. The `grep` user interface conforms to the Unix command-line style. It consists of the command name, here `grep`, options, and the arguments. The first argument is the regular expression delimited by single straight quotes. The next arguments are the files where to search the pattern:

```
grep 'regular expression' file1 file2 ... filen
```

The Unix command:

```
grep 'abc' myFile
```

prints all the lines of file `myFile` containing the string abc and

```
grep 'ab*c' myFile1 myFile2
```

prints all the lines of file `myFile1` and `myFile2` containing the strings ac, abc, $abbc$, $abbbc$, etc.

`grep` had a considerable influence on its followers. Most of them adhere to a comparable syntax. Among the most popular languages featuring regexes now are Perl and Python, Java, and C#. In the following sections, the description of the syntactic features refers to `egrep`, which is a modern version of `grep` available for most operating systems.

2.3.1 Repetition Metacharacters

We saw that the metacharacter `*` expressed a repetition of zero or more characters, as in `ab*c`. Other characters that describe repetitions are the question mark, `?`, the plus, `+`, and the dot, `.` (Table 2.9). The star symbol is also called the closure operator or the Kleene star.

If the pattern to search contains a character that is also a metacharacter, for instance, "?", we need to indicate it to the regex engine using a backslash \ before it. We saw that `abc?` matches ab and abc. The expression `abc\?` matches the string $abc?$. In the same vein, `abc\.` matches the string $abc.$, and `a\*bc` matches $a*bc$. The backslash is also called the escape character. It transforms a metacharacter into a literal symbol. In most regex languages, we must quote characters `.`, `?`, `(`, `)`, `[`, `]`, `{`, `}`, `*`, `+`, `|`, `^`, `$`, and \ to search them literally.

Table 2.9. Repetition metacharacters.

Metachars	Descriptions	Examples
`*`	Matches any number of occurrences of the previous character – zero or more	`ac*e` matches strings `ae`, `ace`, `acce`, `accce`, etc. as in "The aerial acceleration alerted the ace pilot"
`?`	Matches at most one occurrence of the previous characters – zero or one	`ac?e` matches `ae` and `ace` as in "The aerial acceleration alerted the ace pilot"
`+`	Matches one or more occurrences of the previous characters	`ac+e` matches `ace`, `acce`, `accce`, etc. as in as in "The aerial acceleration alerted the ace pilot"
`{n}`	Matches exactly n occurrences of the previous characters	`ac{2}e` matches `acce` as in "The aerial acceleration alerted the ace pilot"
`{n,}`	Matches n or more occurrences of the previous characters	`ac{2,}e` matches `acce`, `accce`, etc.
`{n,m}`	Matches from n to m occurrences of the previous characters	`ac{2,4}e` matches `acce`, `accce`, and `acccce`.
`.`	Matches one occurrence of any characters of the alphabet except the new line character	`a.e` matches `aae`, `aAe`, `abe`, `aBe`, `a1e`, etc. as in "The aerial acceleration alerted the ace pilot"
`.*`	Matches any string of characters and until it encounters a new line character	

2.3.2 The Longest Match

The description of repetition metacharacters in Table 2.9 sometimes makes string matching ambiguous, as with the string `aabbc` and the regex `a+b*`, which has six possible matches: a, aa, ab, aab, abb, and $aabb$. In fact, matching algorithms use two rules that are common to all the regex languages:

1. They match as early as they can in a string.
2. They match as many characters as they can.

Hence, `a+b*` matches $aabb$, which is the longest possible match. The matching strategy of repetition metacharacters is said to be greedy.

In some cases, the greedy strategy is not appropriate. To display the sentence

*They match **as early** and **as many** characters as they can.*

in a Web page with two phrases set in bold, we need specific tags that we will insert in the source file. Using HTML, the language of the Web, the sentence will probably be annotated as

```
They match <b>as early</b> and <b>as many</b>
characters as they can.
```

where and mark respectively the beginning and the end of a phrase set in bold. (We will see annotation frameworks in more detail in Chap. 3.)

A regular expression to search and extract phrases in bold could be:

```
<b>.*</b>
```

Unfortunately, applying this regex to the sentence will match one single string:

as early and as many

which is not what we wanted. In fact, this is not a surprise. As we saw, the regex engine matches as early as it can, i.e., from the first and as many characters as it can up to the second .

A possible solution is to modify the behavior of repetition metacharacters and make them "lazy." They will then consume as few characters as possible. We create the lazy variant of a repetition metacharacter by appending a question mark to it (Table 2.10). The regex

```
<b>.*?</b>
```

will then match the two intended strings,

as early and as many.

Table 2.10. Lazy metacharacters.

Metachars	Descriptions
`*?`	Matches any number of occurrences of the previous character – zero or more
`??`	Matches at most one occurrence of the previous characters – zero or one
`+?`	Matches one or more occurrences of the previous characters
`{n}?`	Matches exactly n occurrences of the previous characters
`{n,}?`	Matches n or more occurrences of the previous characters
`{n,m}?`	Matches from n to m occurrences of the previous characters

2.3.3 Character Classes

We saw that the dot, `.`, represents any character of the alphabet. It is possible to define smaller subsets or **classes**. A list of characters between square brackets `[...]` matches any character contained in the list. `[abc]` means one occurrence of either `a`, `b`, or `c`. `[ABCDEFGHIJKLMNOPQRSTUVWXYZ]` means one uppercase unaccented letter, and `[0123456789]` means one digit. We can concatenate character classes, literal characters, and metacharacters, as in the expressions `[0123456789]+` and

`[0123456789]+\.[0123456789]+`, that match respectively integers and decimal numbers.

Character classes are useful to search patterns with spelling differences, such as `[Cc]omputer [Ss]cience`, which matches four different strings:

Computer Science
Computer science
computer Science
computer science

We can define the complement of a character class, that is, the characters of the alphabet that are not member of the class, using the caret symbol, `^`, as the first symbol inside the angle brackets. `[^a]` means any character that is not an *a*. `[^0123456789]` means any character that is not a digit. The expression `[^ABCD]*` means any string that does not contain *A*, *B*, *C*, or *D*. The caret must be the first character after the brackets. The expression `[a^b]` matches either *a*, ^, or *b*.

Inside angle brackets, we can also specify ranges using the dash character `-`. The expression `[1-4]` means any of the digits *1*, *2*, *3*, or *4*, and `a[1-4]b` matches *a1b*, *a2b*, *a3c*, or *a4b*. The expression `[a-zàâäæçéèêëîïôöœßùûüÿ]` matches any lowercase accented or unaccented letter of French and German. If we want to search the dash character itself, we need to quote it as `\-`. The expression `[1\-4]` means any of the characters *1*, -, or *4*.

Most regex languages have also predefined classes. Table 2.11 lists some useful ones. Some classes may be specific to one regex language. In case of doubt, refer to the corresponding manual.

2.3.4 Nonprintable Symbols or Positions

Some metacharacters match positions and nonprintable symbols. Positions or **anchors** enable one to search a pattern with a specific location in a text. They encode the start and end of a line, using respectively the caret, `^`, and the dollar, `$`.

The expression `^Chapter` matches lines beginning with *Chapter* and `[0-9]+$` matches lines ending with a number. We can combine both in `^Chapter [0-9]+$` that matches lines consisting only of the *Chapter* word and a number as *Chapter 3*, for example.

The command line

```
egrep '^[aeiou]+$' myFile
```

matches lines of `myFile` containing only vowels.

Similarly, metacharacters `\<` and `\>` match the start and end of a word. The expression `\<ace` matches *aces* and *acetylene* but not *place*. Conversely, `ace\>` matches *place* but neither *aces* nor *acetylene*. The expression `\<act\>` matches exactly the word *act* and not *react* or *acted*. Table 2.12 summarizes anchors and some nonprintable characters.

In Perl, word boundaries are indicated by `\b` instead of `\<` and `\>`, as in `\bact\b`.

Table 2.11. Predefined character classes.

Expressions	Descriptions	Examples
\d	Any digit. Equivalent to [0-9]	A\dC matches A0C, A1C, A2C, A3C etc.
\D	Any nondigit. Equivalent to [^0-9]	
\w	Any word character: letter, digit, or underscore. Equivalent to [a-zA-Z0-9_]	1\w2 matches 1a2, 1A2, 1b2, 1B2, etc
\W	Any nonword character. Equivalent to [^\w]	
\s	Any white space character: space, tabulation, new line, form feed, carriage return, or backspace.	
\S	Any nonwhite space character. Equivalent to [^\s]	
[:alpha:]	Any alphabetic character. It includes accented characters	1[:alpha:]2 matches 1a2, 1A2, 1b2, 1B2, etc.
[:digit:]	Any digit	A[:digit:]C matches A0C, A1C, A2C, A3C etc.
[:upper:]	Any uppercase character. It includes accented characters	A[:upper:]C matches AAC, ABC, ACC, ADC etc.
[:lower:]	Any lowercase character. It includes accented characters	A[:lower:]C matches AaC, AbC, AcC, AdC etc.

Table 2.12. Some metacharacters matching nonprintable characters.

Metachars	Descriptions	Examples
^	Matches the start of a line	^ab*c matches ac, abc, abbc, abbbc, etc. when they are located at the beginning of a new line
$	Matches the end of a line	ab?c$ matches ac and abc when they are located at the end of a line
\<	Matches the start of a word	\<abc matches abcd but not dabc
\>	Matches the end of a word	bcd\> matches abcd but not abcde
\n	Matches a new line	a\nb matches a b
\t	Matches a tabulation	–
\r	Matches the carriage return character	–
\f	Matches the form feed character	–
\e	Matches the escape character	–
\a	Matches the bell character	–

2.3.5 Union and Boolean Operators

We reviewed the basic constructs to write regular expressions. A powerful feature is that we can also combine expressions with operators, as with automata. Using a mathematical term, we say that they define an algebra. Using a simpler analogy, this means that we can arrange regular expressions just like arithmetic expressions. This greatly eases the design of complex expressions and makes them very versatile.

Regex languages use three main operators. Two of them are already familiar to us. The first one is the Kleene star or closure, denoted `*`. The second one is the concatenation, which is usually not represented. It is implicit in strings like `abc`, which is the concatenation of characters a, b, and c. To concatenate the word *computer*, a space symbol, and *science*, we just write them in a row: `computer science`.

The third operation is the union and is denoted "|". The expression `a|b` means either a or b. We saw that the regular expression `[Cc]omputer [Ss]cience` could match four strings. We can rewrite an equivalent expression using the union operator: `Computer Science|Computer science|computer Science|computer science`. A union is also called an alternation because the corresponding expression can match any of the alternatives, here four.

2.3.6 Operator Combination and Precedence

Regular expressions and operators are grouped using parentheses. If we omit them, expressions are governed by rules of precedence and associativity. The expression `a|bc` matches the strings a and bc because the concatenation operator takes precedence over the union. In other words, the concatenation binds the characters stronger than the union. If we want an expression that matches the strings ac and bc, we need parentheses `(a|b)c`.

Let us examine another example of precedence. We rewrote the expression `[Cc]omputer [Ss]cience` using a union of four strings. Since the difference between expressions lies in the first letters only, we can try to revise this union into something more compact. The character class `[Cc]` is equivalent to the alternation `C|c`, which matches either C or c. A tentative expression could then be `C|computer S|science`. But it would not match the desired strings; it would find occurrences of either C, *computer S*, or *science* because of the operator precedence. We need parentheses to group the alternations `(C|c)omputer (S|s)cience` and thus match the four intended strings.

The order of precedence of the three main operators union, concatenation, and closure is as follows:

1. closure and other repetition operator (highest)
2. concatenation, line and word boundaries
3. union (lowest)

This entails that `abc*` describes the set ab, abc, $abcc$, $abccc$, etc. To repeat the pattern abc, we need parentheses. And the expression `(abc)*` corresponds to abc, $abcabc$, $abcabcabc$, etc.

2.4 Programming with Regular Expressions

2.4.1 Perl

`grep` and `egrep` are tools to search patterns in texts. If we want to use them for more elaborate text processing such as translating characters, substituting words, counting them, we need a full-fledged programming language, for example, Perl, Python, AWK, and Java with its `java.util.regex` package. They enable the design of powerful regular expressions and at the same time, they are complete programming languages. This section intends to give you a glimpse of Perl programming. We discuss features of Perl in this chapter and the next one. Further references include Wall et al. (2000) and Schwartz and Phoenix (2001).

2.4.2 Matching

Perl has constructs similar to those of the C language. It has analogous control flow statements, and the assignment operator is denoted =. However, variables begin with a dollar sign and are not typed. Comments start with the # symbol. The short program

```
# A first program
$integer = 30;
$pattern = "My string";
print $integer, " ", $pattern, "\n";
```

prints the line

```
30 My string
```

We run it with the command:

```
perl -w program.pl
```

where the option `-w` asks Perl to check syntax errors.

The next program reads the input line and searches the expression `ab*c`. If it finds the expression, it prints the line:

```
while ($line = <>) {
  if ($line =~ m/ab*c/) {
    print $line;
  }
}
```

The program uses repeat and conditional statements. The symbol `<>` designates the standard input, and the instruction `$line = <>` assigns the current line from the input to the `$line` variable. The `while` instruction reads all the lines until it encounters an end of file. The `m/.../` instruction delimits the regular expression to match, and the `=~` operator instructs Perl to search it in the `$line` variable. If the expression matches a string in `$line`, the `=~` operator returns true, or false otherwise. The `if` instruction tells the program to print the input when it contains the pattern. We run the program to search the file `file_name` with the command:

```
perl -w program.pl file_name
```

The match operator supports a set of options also called modifiers. Their syntax is `m/regex/modifiers`. Useful modifiers are

- Case insensitive: `i`. The instruction `m/regex/i` searches `regex` in the target string regardless of its case.
- Multiple lines: `m`. By default, the anchors `^` and `$` match the start and the end of the input string. The instruction `m/regex/m` considers the input string as multiple lines separated by new line characters, where the anchors `^` and `$` match the start and the end of any line in the string.
- Single line: `s`. Normally, a dot symbol "`.`" does not match new line characters. The `/s` modifier makes a dot in the instruction `m/regex/s` match any character including new lines.

Modifiers can be grouped in any order as in `m/regex/im`, for instance, or `m/regex/sm`, where a dot in `regex` matches any character and the anchors `^` and `$` match just after and before new line characters.

2.4.3 Substitutions

One of the powerful features of Perl is pattern substitution. It uses a construct similar to the match instruction: `s/regex/replacement/`. The instruction

```
$line =~ s/regex/replacement/
```

matches the first occurrence of `regex` and replaces it by `replacement` in the `$line` variable. If we want to replace all the occurrences of a pattern, we use the `g` modifier, where `g` stands for globally:

```
$line =~ s/regex/replacement/g
```

We shall write a program to replace the occurrences of `ab*c` by `ABC` in a file and print them. We read all the lines of the input. We use the instruction `m/ab*c/` to check whether they match the regular expression `ab*c`. We then print the old line and we substitute the matched pattern using the construct `s/ab*c/ABC/`:

```
while ($line = <>) {
  if ($line =~ m/ab*c/) {
    print "Old: ", $line;
    $line =~ s/ab*c/ABC/g;
    print "New: ", $line;
  }
}
```

2.4.4 Translating Characters

The instruction `tr/search_list/replacement_list/` replaces all the occurrences of the characters in `search_list` by the corresponding character in `replacement_list`. The instruction `tr/ABC/abc/` replaces the occurrences of *A*, *B*, and *C* by *a*, *b*, and *c*, respectively. The string

```
AbCdEfGhIjKlMnOpQrStUvWxYzÉö
```

results in

```
abcdEfGhIjKlMnOpQrStUvWxYzÉö
```

The hyphen specifies a character range, as in the instruction

```
$line =~ tr/A-Z/a-z/;
```

which converts the uppercase characters to their lowercase equivalents. The instruction `tr` has useful modifiers:

- `d` deletes any characters of the search list that are not found in the replacement list.
- `c` translates characters that belong to the complement of the search list.
- `s` reduces – squeezes, squashes – sequences of characters translated to an identical character to a single instance.

The instruction

```
$line =~ tr/AEIOUaeiou//d;
```

deletes all the vowels in `$line` and

```
$line =~ tr/AEIOUaeiou/\$/cs;
```

replaces all nonvowel characters by a `$` sign. The contiguous sequences of translated dollar signs are reduced to a single sign.

2.4.5 String Operators

Perl operators are similar to those of the C and Java languages. They are summarized in Table 2.13. The string operators are notable differences. They enable us to concatenate and compare strings.

The Boolean operators `eq` (equal) and `ne` (not equal) compare two strings. The dot is the concatenation operator:

```
$string1 = "abc";
$string 2 = "def";
$string3 = $string1 . $string2;
print $string3;
#prints abcdef
```

As with the C and Java operators, the shorthand notation `$var1 .= $var2` is equivalent to `$var1 = $var1 . $var2`. The following program reads the content of the input line by line, concatenates it in the `$text` variable, and prints it:

```
while ($line = <>) {
  $text .= $line;
}
print $text;
```

Table 2.13. Summary of the main Perl operators.

Unary operators	!	Logical not
	+ and -	Arithmetic plus sign and negation
Binding operators	=~	Returns true in case of match success
	!~	Returns false in case of match success
Arithmetic operators	* and /	Multiplication and division
	+ and -	Addition and subtraction
String operator	.	String concatenation
Arithmetic comparison operators	> and <	Greater than and less than
	>= and <=	Greater than or equal and less than or equal
	== and !=	Equal and not equal
String comparison operators	ge and le	Greater than and less than
	gt and lt	Greater than or equal and less than or equal
	eq and ne	Equal and not equal
Logical operators	&&	Logical and
	||	Logical or

2.4.6 Back References

It is sometimes useful to keep a reference to matched patterns or parts of them. Let us imagine that we want to find a sequence of three identical characters, which corresponds to matching a character and checking if the next two characters are identical to the first character. To do this, we first tell Perl to remember the matched pattern and we put parentheses around it. It creates a buffer to hold the pattern and we refer back to it by the sequence `\1`. The instruction `s/(.)\1\1/\*\*\*/g` replaces these sequences by three stars.

Perl can create as many buffers as we need. It allocates a new one when it encounters a left parenthesis and refers it back by references \1, \2, \3, etc. The first pair of parentheses corresponds to \1, the second pair to \2, the third to \3, etc. Outside the match expression the \<digit> reference is denoted by $<digit>: $1, $2, $3, etc. As an example, the next program captures occurrences of money amounts in dollars. It prints the dollars and cents:

```
while ($line = <>) {
  while ($line =~ m/\$ *([0-9]+)\.?([0-9]*)/g) {
    print "Dollars: ", $1, " Cents: ", $2, "\n";
  }
}
```

2.5 Finding Concordances

2.5.1 Concordances in Prolog

Concordances of a word, an expression, or more generally any string in a corpus are easy to obtain with Prolog. Let us suppose that the corpus is represented as one single big string: a list of characters. Concordancing simply consists in matching the pattern we are searching as a substring of the whole list. There is no need to consider the corpus structure, that is, whether it is made of blanks, words, sentences, or paragraphs.

We implement the search with two auxiliary predicates: `prefix(+List, +Span, -Prefix)` that extracts the prefix of a list with up to `Span` characters, and `prepend(+List, +Span, -PrependedList)` that adds `Span` variables onto the beginning of a list.

Now let us write the `concordance/4` predicate. It finds `Pattern` in `List` and returns the first `Line` where it occurs. `Span` is the window size, for example, 15 characters to the left and to the right, within which `Pattern` will be displayed. We first prepend `Pattern` with `Span` variables before it to match the pattern and its right context. We find it with a combination of two `append/3` calls; then we use `prefix/3` to extract up to `Span` characters after it.

```
% concordance(+Pattern, +List, +Span, -Line)
% finds Pattern in List and displays the Line
% where it appears within Span characters
% surrounding it.

concordance(Pattern, List, Span, Line) :-
  name(Pattern, LPattern),
  prepend(LPattern, Span, LeftPattern),
  append(_, Rest, List),
  append(LeftPattern, End, Rest),
  prefix(End, Span, Suffix),
```

```
  append(LeftPattern, Suffix, LLine),
  name(Line, LLine).

% prefix(+List, +Span, -Prefix) extracts the prefix
% of List with up to Span characters.
% The second rule is to check the case where there
% are less than Span character in List.

prefix(List, Span, Prefix) :-
  append(Prefix, _, List),
  length(Prefix, Span),
  !.
prefix(Prefix, Span, Prefix) :-
  length(Prefix, L),
  L < Span.

% prepend(+List. +Span, -Prefix) adds Span variables
% to the beginning of List.

prepend(Pattern, Span, List) :-
  prepend(Pattern, Span, Pattern, List).

prepend(_, 0, List, List) :- !.
prepend(Pattern, Span, List, FList) :-
  Span1 is Span - 1,
  prepend(Pattern, Span1, [X | List], FList).
```

Let us apply this program to retrieve the concordances of *Helen* in the *Iliad*. We make `concordance/4` backtrack until all the occurrences have been found:

```
?- read_file('iliad.txt', L), concordance('Helen', L,
20, C), write(C),nl, fail.

ry of still keeping Helen, for whose sake so
ry of still keeping Helen, for whose sake so
red for the sake of Helen. Nevertheless, if a
red for the sake of Helen. The men of Pylos
 in their midst for Helen and all her wealth.
he midst of you for Helen and all her wealth.
nwhile Iris went to Helen in the form of her
ke the goddess, and Helen's heart yearned aft
wood. When they saw Helen coming towards the
"   "Sir," answered Helen, "father of my husb
...
No
```

Because the pattern is prepended with exactly `Span` variables, the concordance program will not examine the first `Span` characters of the file. This means that it will not find a possible pattern in this sublist. In our example above, the program finds all the occurrences of *Helen* except the ones that could occur in the first 15 characters of the text. This is easily corrected in the program and is left as an exercise.

2.5.2 Concordances in Perl

Arrays in Perl. Writing a basic concordance program is also easy in Perl. However, to be convenient, the program must be able to read parameters from the command line – the file name, the pattern to search, and the span size of the concordance – as in

```
perl -w concordance.pl corpus.txt my_word 15
```

These arguments are passed to Perl by the operating system under the form of an array. Before writing the program, we introduce this feature now.

Arrays in Perl are data structures that can hold any number of elements of any type. Their name begins with an at sign, `@`, for example, `@array`. Each element has a position where the programmer can store and read data using the position index.

An array grows or shrinks automatically when elements are appended, inserted, or deleted. Perl manages the memory without any intervention from the programmer. Here are some examples of arrays:

```
@array1 = ();         # The empty array
@array2 = (1, 2, 3); # Array containing 1, 2, and 3

$var1 = 3.14;
$var2 = "my string";
@array3 = (1, $var1, "Prolog", $var2);
# Array containing four elements of different type

@array4 = (@array2,@array3);
#Same as (1, 2, 3, 1, 3.14, "Prolog", "my string")
```

Reading or assigning a value to a position of the array is done using its index between square brackets starting from 0:

```
print $array2[1];  # prints 2
```

If an element is assigned to a position that did not exist before, Perl grows the array to store it. The positions in-between are not initialized. They hold the value `undef`:

```
$array4[10] = 10;
print $array4[10]; # prints 10
print $array4[9];
# prints a message telling it is undefined
```

The existence of a variable can be tested using the `defined` Boolean function as in:

```
if (defined($array4[9])) {
  print "yes", "\n";
} else {
  print "no", "\n";
}
```

If an `undef` value is used as a number, it is considered to be a zero. The next two lines print 1.

```
$array4[9]++;
print $array4[9];
```

The variable `$#array` is the index of the last element of the array. It can be assigned to grow or shrink the array:

```
$length4 = $#array4;
print $length4;   # prints 10
print $#array2;   # prints 2
$#array4 = 5;     # shrinks the array to 6 elements.
                  # Other elements are lost.
print $array4[10];
# prints a message telling it is undefined
$#array2 = 10;    # extends the array to 11 elements.
                  # Indices 3..10 are undefined.
```

You can also assign a complete array to an array and an array to a list of variables as in:

```
@array5 = @array2;
($v1, $v2, $v3) = @array2;
```

where `@array5` contains a copy of `@array2`, and `$v1`, `$v2`, `$v3` contain respectively 1, 2, and 3.

Printing Concordances in Perl. Now let us write a concordance program modified from Cooper (1999). First, we read the command line arguments: the file name, the pattern to search, and the span size. They are stored in the reserved variable `@ARGV`. We open the file using the `open` function, which assigns the stream to the `FILE` identifier. If `open` fails, the program exits using `die` and prints a message to inform us that it could not open the file.

The notation `<FILE>` designates the input stream, which is assigned to the `$line` variable. We read all the text and we assign it to the `$text` variable. To allow matching across spaces, tabulations, and new lines, we replace spaces in the regular expression `$pattern` representing the pattern to search by the space metacharacter `\s`. We also replace the new lines in the text by a space.

Finally, we use a `while` loop to match the pattern with `$width` characters to the left and to the right. The `/g` modifier enables the `m/.../` instruction to match a pattern and to start a new search from its current position – where the previous match ended. When `m/.../g` fails to match, the start position is reset to the beginning of the string. We create a back reference by setting parentheses around the regular expression to remember the matched pattern and we print it.

```
($file_name, $pattern, $width) = @ARGV;
open(FILE, "$file_name") ||
      die "Could not open file $file_name.";
while ($line = <FILE>) {
  $text .= $line;
}
$pattern =~ s/ /\\s/g;
      # spaces match tabs and new lines
$text =~ s/\n/ /g;
      # new lines are replaced by spaces
while ($text =~ m/(.{0,$width}$pattern.{0,$width})/g){
      # matches the pattern with 0..width
      #to the right and left
  print "$1\n"; #$1 contains the match
}
```

Now let us run the command:

```
perl -w concordance.pl odyssey.txt Penelope 20

itors of his mother Penelope, who persist in eat
ying out yet, while Penelope has such a fine son
 upon the Achaeans. Penelope, daughter of Icariu
d of Ulysses and of Penelope in your veins I see
long-suffering wife Penelope, and his son Telema
It was not long ere Penelope came to know what t
reshold of her room Penelope said: "Medon, what
```

2.6 Approximate String Matching

So far, we have used regular expressions to match exact patterns. However, in many applications, such as in spell checkers, we need to extend the match span to search a set of related patterns or strings. In this section, we review techniques to carry out approximate or inexact string matching.

2.6.1 Edit Operations

A common method to create a set of related strings is to apply a sequence of edit operations that transforms a source string s into a target string t. The operations are

carried out from left to right using two pointers that mark the position of the next character to edit in both strings:

- The copy operation is the simplest. It copies the current character of the source string to the target string. Evidently, the repetition of copy operations produces equal source and target strings.
- Substitution replaces one character from the source string by a new character in the target string. The pointers are incremented by one in both the source and target strings.
- Insertion inserts a new character in the target string. The pointer in the target string is incremented by one, but the pointer in the source string is not.
- Deletion deletes the current character in the target string, i.e., the current character is not copied in the target string. The pointer in the source string is incremented by one, but the pointer in the target string is not.
- Reversal (or transposition) copies two adjacent characters of the source string and transposes them in the target string. The pointers are incremented by two characters.

Kernighan et al. (1990) illustrate these operations with the misspelled word *acress* and its possible corrections (Table 2.14).

Table 2.14. Typographical errors (typos) and corrections. Strings differ by one operation. The correction is the source and the typo is the target. Unless specified, other operations are just copies. After Kernighan et al. (1990).

Typo	Correction	Source	Target	Position	Operation
acress	actress	–	t	2	Deletion
acress	cress	a	–	0	Insertion
acress	caress	ac	ca	0	Transposition
acress	access	r	c	2	Substitution
acress	across	e	o	3	Substitution
acress	acres	s	–	4	Insertion
acress	acres	s	–	5	Insertion

If we allow only one edit operation on a source string of length n, and if we consider an alphabet of 26 unaccented letters, the deletion will generate n new strings; the insertion, $(n + 1) \times 26$ strings; the substitution, $n \times 25$; and the transposition, $n - 1$ new strings.

2.6.2 Minimum Edit Distance

Complementary to edit operations, edit distances measure the similarity between strings. They assign a cost to each edit operation, usually 0 to copies and 1 to deletions and insertions. Substitutions and transpositions correspond both to an insertion and a deletion. We can derive from this that they each have a cost of 2. Edit distances

tell how far a source string is from a target string: the lower the distance, the closer the strings.

Given a set of edit operations, the minimum edit distance is the operation sequence that has the minimal cost needed to transform the source string into the target string. If we restrict the operations to copy/substitute, insert, and delete, we can represent the edit operations using a table, where the distance at a certain position in the table is derived from distances in adjacent positions already computed. This is expressed by the formula:

$$edit\_distance(i,j) = \min \begin{pmatrix} edit\_distance(i-1,j) + del\_cost \\ edit\_distance(i-1,j-1) + subst\_cost \\ edit\_distance(i,j-1) + ins\_cost \end{pmatrix}.$$

The boundary conditions for the first row and the first column correspond to a sequence of deletions and of insertions. They are defined as $edit\_distance(i,0) = i$ and $edit\_distance(0,j) = j$.

We compute the cell values as a walk through the table from the beginning of the strings at the bottom left corner, and we proceed upward and rightward to fill adjacent cells from those where the value is already known. Arrows in Fig. 2.10 represent the three edit operations, and Table 2.15 shows the distances to transform *language* into *lineage*. The value of the minimum edit distance is 5 and is shown at the upper right corner of the table.

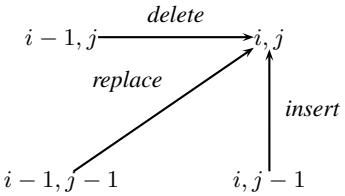

Fig. 2.10. Edit operations.

Table 2.15. Distances between *language* and *lineage*.

e	7	6	5	6	5	6	7	6	**5**
g	6	5	4	5	4	5	6	5	6
a	5	4	3	4	5	6	5	6	7
e	4	3	4	3	4	5	6	7	6
n	3	2	3	2	3	4	5	6	7
i	2	1	2	3	4	5	6	7	8
l	1	0	1	2	3	4	5	6	7
Start	0	1	2	3	4	5	6	7	8
–	**Start**	**l**	**a**	**n**	**g**	**u**	**a**	**g**	**e**

The minimum edit distance algorithm is part of the **dynamic programming** techniques. Their principles are relatively simple. They use a table to represent data, and they solve a problem at a certain point by combining solutions to subproblems. Dynamic programming is a generic term that covers a set of widely used methods in optimization.

We implement the minimum edit distance in Perl. We introduce the `length` function to compute the length of the source and target, and we use `split(//, $string)` to convert a string into an array of characters. The instruction

```
@array = split(regex, $string)
```

breaks up the `$string` variable as many times as `regex` matches in `$string`. The `regex` expression acts as a separator, and the string pieces are assigned sequentially to `@array`. In the program, `regex` is reduced to nothing and assigns all the characters `$string` as elements of `@array`.

```
($source, $target) = @ARGV;
$length_s = length($source);
$length_t = length($target);
# Initialize first row and column
for ($i = 0; $i <= $length_s; $i++) {
  $table[$i][0] = $i;
}
for ($j = 0; $j <= $length_t; $j++) {
  $table[0][$j] = $j;
}
# Get the characters. Start index is 0
@source = split(//, $source);
@target = split(//, $target);
# Fills the table.
# Start index of rows and columns is 1
for ($i = 1; $i <= $length_s; $i++) {
  for ($j = 1; $j <= $length_t; $j++) {
    # Is it a copy or a substitution?
    $cost = ($source[$i-1] eq $target[$j-1]) ? 0: 2;
    # Computes the minimum
    $min = $table[$i-1][$j-1] + $cost;
    if ($min > $table[$i][$j-1] + 1) {
      $min = $table[$i][$j-1] + 1;
    }
    if ($min > $table[$i-1][$j] + 1) {
      $min = $table[$i-1][$j] + 1;
    }
    $table[$i][$j] = $min;
  }
}
```

```
print "Minimum distance: ",
$table[$length_s][$length_t], "\n";
```

2.6.3 Searching Edits in Prolog

Once we have filled the table, we can search the operation sequences that correspond to the minimum edit distance. Such a sequence is also called an **alignment**.

The depth-first strategy is an economical way to traverse a search space. It is easy to implement in Prolog and has low memory requirements. The problem with it is that it blindly selects the paths to follow and can explore very deep nodes while ignoring shallow ones. To avoid this, we apply a variation of the depth-first search where we fix the depth in advance to the minimum edit distance. We assign it in the call parameter `Cost` of `edit_distance/4`.

The code of the depth-limited search is similar to the depth-first program (see Appendix A). We add a counter in the recursive case that represents the current search depth and we increment it until we have reached the depth limit. We compute each individual edit operation and its cost with the `edit_operation/6` predicate.

```
% edit_distance(+Source, +Target, -Edits, +Cost).
edit_distance(Source, Target, Edits, Cost) :-
  edit_distance(Source, Target, Edits, 0, Cost).

edit_distance([], [], [], Cost, Cost).
edit_distance(Source, Target, [EditOp | Edits], Cost,
     FinalCost) :-
  edit_operation(Source, Target, NewSource,
       NewTarget, EditOp, CostOp),
  Cost1 is Cost + CostOp,
  edit_distance(NewSource, NewTarget, Edits, Cost1,
       FinalCost).

% edit_operation carries out one edit operation
% between a source string and a target string.
edit_operation([Char | Source], [Char | Target],
     Source, Target, ident, 0).
edit_operation([SChar | Source], [TChar | Target],
     Source, Target, sub(SChar,TChar), 2) :-
  SChar \= TChar.
edit_operation([SChar | Source], Target, Source,
     Target, del(SChar), 1).
edit_operation(Source, [TChar | Target], Source,
     Target, ins(TChar), 1).
```

Using backtracking, Prolog finds all the alignments. We obtain with the minimum distance of 5:

```
?- edit_distance([l,a,n,g,u,a,g,e], [l,i,n,e,a,g,e],
 E, 5).

E = [ident, sub(a, i), ident, sub(g, e), del(u),
     ident, ident, ident] ;

E = [ident, sub(a, i), ident, del(g), sub(u, e),
     ident, ident, ident] ;

E = [ident, sub(a, i), ident, del(g), del(u), ins(e),
     ident, ident, ident]
...
```

with 15 possible alignments in total. Figure 2.6.3 shows the first and third ones.

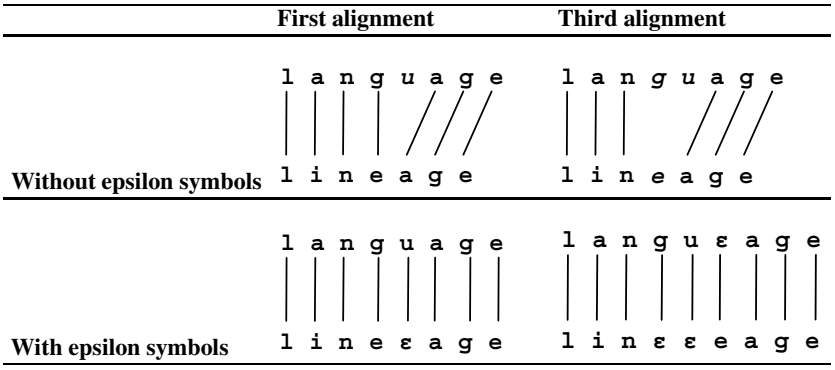

Fig. 2.11. Alignments of *lineage* and *language*. The figure contains two possible representations of them. In the upper row, the deletions in the source string are in italics, as are the insertions in the target string. The lower row shows a synchronized alignment, where deletions in the source string as well as the insertions in the target string are aligned with epsilon symbols (null symbols).

We can apply this Prolog search program alone to find the edit distance. We avoid going an infinite path with an iterative deepening. We start with an edit distance of 0 (the `Cost` parameter) and we increment it – 1, 2, 3, 4 – until we find the minimum edit distance. The first searches will fail, and the first one that succeeds corresponds to the minimum distance.

2.7 Further Reading

Corpora are now easy to obtain. Organizations such as the Linguistic Data Consortium and ELRA collect and distribute texts in many languages. Although not

widely cited, the first fiction corpus with more than 100 million words was probably FranText, which helped write the *Trésor de la langue française* (Imbs 1971–1994). Other early corpora include the Bank of English, which contributed to the *Collins COBUILD Dictionary* (Sinclair 1987).

Text and corpus analysis are an active focus of research in computational linguistics. They include the description of word distributions that were theorized at the beginning of the 20th century by Bloomfield and followers such as Harris (1962). Paradoxically, natural language processing conducted by computer scientists largely ignored corpora until the 1990s, when it rediscovered techniques routinely used in humanities. For a short history, see Zampolli (2003).

Roche and Schabes (1997, Chap. 1) is a concise and clear introduction to automata theory. It makes an extensive use of mathematical notations, however. Hopcroft et al. (2001) is a standard and comprehensive textbook on automata and regular expressions. Friedl (2002) is a thorough presentation of regular expressions oriented toward applications and programming techniques.

Although the idea of automata underlies some mathematical theories of the 19th century such as those of Markov, Gödel, or Turing, Kleene (1956) was first to give a formal definition. He also proved the equivalence between regular expressions and FSA. Thompson (1968) was the first to implement a widely used editor embedding a regular expression tool: Global/Regular Expression/Print, better known as `grep`.

There are several FSA toolkits available from the Internet. The FSA utilities (van Noord and Gerdemann 2001) is a Prolog package to manipulate regular expressions, automata, and transducers (odur.let.rug.nl/~vannoord/Fsa/). The FSM library (Mohri et al. 1998) is another set of tools (www.research.att.com/sw/tools/fsm/). Both include rational operations – union, concatenation, closure, reversal – and equivalence transformation – ε-elimination, determinization, and minimization.

Exercises

2.1. Implement the automaton in Fig. 2.5.

2.2. Implement a Prolog program to automatically construct an automaton to search a given input string.

2.3. Write a regular expression that finds occurrences of *honour* and *honor* in a text.

2.4. Write a regular expression that finds lines containing all the vowels *a*, *e*, *i*, *o*, *u*, in that order.

2.5. Write a regular expression that finds lines consisting only of letters *a*, *b*, or *c*.

2.6. List the strings generated by the expressions:

```
(ab)*c
(a.)*c
(a|b)*
a|b*|(a|b)*a
a|bc*d
```

2.7. Complement the Prolog concordance program to sort the lines according to words appearing on the right of the string to search.

2.8. Write the iterative deepening search in Prolog to find the minimum edit distance.