Formal Language Theory for Natural Language Processing

Shuly Wintner

Department of Computer Science
University of Haifa
31905 Haifa, Israel

© 2001 by Shuly Wintner

Contents

1 Set theory **1**
- 1.1 Sets . . . 1
- 1.2 Relations . . . 3
- 1.3 Strings . . . 5
- 1.4 Languages . . . 6

2 Regular languages **9**
- 2.1 Regular expressions . . . 9
- 2.2 Properties of regular languages . . . 11
- 2.3 Finite-state automata . . . 12
- 2.4 Minimization and determinization . . . 15
- 2.5 Operations on finite-state automata . . . 17
- 2.6 Applications of finite-state automata in natural language processing . . . 19
- 2.7 Regular relations . . . 20
- 2.8 Finite-state transducers . . . 21
- 2.9 Properties of regular relations . . . 23

3 Context-free grammars and languages **27**
- 3.1 Where regular languages fail . . . 27
- 3.2 Grammars . . . 28
- 3.3 Derivation . . . 29
- 3.4 Derivation trees . . . 31
- 3.5 Expressiveness . . . 33
- 3.6 Linguistic examples . . . 36
- 3.7 Formal properties of context-free languages . . . 38

4 The Chomsky hierarchy **39**
- 4.1 A hierarchy of language classes . . . 39
- 4.2 The location of natural languages in the hierarchy . . . 40
- 4.3 Weak and strong generative capacity . . . 43

© 2001 by Shuly Wintner

Preface

This is the reader for the course *Formal Language Theory for Natural Language Processing*, taught as part of ESSLLI 2001, the 13th Summer School in Logic, Language and Information. This course is a mild introduction to formal language theory for students with little or no background in formal systems. The motivation is natural language processing, and the presentation is geared towards NLP applications, with extensive linguistically motivated examples. Still, mathematical rigor is not compromised, and students are expected to have a formal grasp of the material by the end of the course.

The topics to be covered include: set theory; regular languages and regular expressions; languages vs. computational machinery; finite state automata; finite state transducers; context free grammars and languages; the Chomsky hierarchy; weak and strong generative capacity; and grammar equivalence.

As this is a new manuscript, I would appreciate feedback about its suitability, clarity, correctness and usefulness, as well as any errors or typos.

Shuly Wintner
Haifa, October 2001

© 2001 by Shuly Wintner

Chapter 1

Set theory

1.1 Sets

A *set* is an abstract, unordered collection of distinct objects, called *members* or *elements* of the set. When some element x is a member of a set A, we write $x \in A$.

Example 1.1 Sets
The set of vowels of the English alphabet is $\{a, e, i, o, u\}$. Its members are *a, e, i, o* and *u*. The set of articles in German is $\{ein, eine, einer, einem, einen, eines, der, die, das, dem, den, des\}$.

Sets can consist of elements that share no other property than being members of the same set: for example, the set containing the letter "a" and the number 17 is a perfectly valid set.

Sets can be either finite or infinite: the set of letters in the English alphabet is finite. The set of sentences in the English language is infinite, as is the set of odd natural numbers. When a set has exactly one element it is called a *singleton*. When a set has no members it is called an *empty set*. This is a very special set: there is exactly one, unique empty set, and while it might seem weird, it is a very useful notion. For example, it is a good representation for the set of colorless green objects.

There are two main ways to specify sets: by listing all their members or by specifying a property that all their members have (a third way, specifying a set of rules which generate all the elements of a set, will be presented later on in this text). Listing all its elements is the obvious way to specify a set: $\{a, e, i, o, u\}$. Note the use of curly brackets '$\{\ \ \}$' around the elements. Since sets are unordered, the order of elements in some specification of a set is irrelevant, and hence the following sets are identical: $\{a, e, i, o, u\} = \{u, i, e, o, a\}$. Since sets are collections of distinct elements, repetition of an element does not change the set (as one is only interested in whether an element is a member of a set, and multiple occurrences do not change this status): $\{a, e, i, o, u\} = \{a, a, e, i, o, o, o, u\}$. The concept of identity here is that two sets are equal if and only if they have exactly the same members. This also explains why there is a unique empty set: its identity is fully determined by the absence of its members. We use a special symbol, '$\emptyset$' to denote the empty set.

Listing all the elements of some set is a useful way to specify the set, but only when the set is finite. For infinite sets, some other method of specification is required. The predicate notation for specifying sets consists in specifying a predicate that must hold for all and only the members of the set. The notation uses curly brackets, some variable (say, 'x') followed by a vertical bar (which should be read as "such that") and then the predicate.

Example 1.2 Predicate notation for specifying sets

The set of vowels of the English alphabet can be specified as $\{x \mid x \textit{ is a vowel of the English alphabet}\}$ (read: "the set of all x's such that x is a vowel of the English alphabet"). Other sets specified similarly are:

$$\{y \mid y \textit{ is a book on syntax}\}$$

$$\{z \mid z \textit{ is a book on syntax and Chomsky wrote } z\}$$

$$\{x \mid x \textit{ is a word in English and the number of letters in } x \textit{ is 1}\}$$

The above examples all specify finite sets. The following are infinite:

$$\{z \mid z \textit{ is an even number}\}$$

$$\{y \mid y \textit{ is a grammatical English sentence}\}$$

When a set A is finite, the number of its members is called its *cardinality* and is denoted $|A|$. The cardinality of a finite set is a natural number. Infinite sets also have cardinalities, but they are not the number of their elements and are not given by natural numbers.

We have already seen one relation on sets, namely equality. Other relations are useful; in particular, we say that a set A is a *subset* of a set B if and only if every element of A is also an element of B (which might optionally contain additional elements). We denote such a relation by $A \subseteq B$. We say that A *is contained in* B, or that B *contains* A.

Example 1.3 Set inclusion

$$\begin{array}{rcl}
\emptyset & \subseteq & \{a, e, o\} \\
\{a, e, o\} & \subseteq & \{x \mid x \textit{ is a vowel of the English alphabet}\} \\
\{a, e, i, o, u\} & \subseteq & \{x \mid x \textit{ is a vowel of the English alphabet}\} \\
\{x \mid x \textit{ is a vowel of the English alphabet}\} & \subseteq & \{x \mid x \textit{ is a letter of the English alphabet}\}
\end{array}$$

Note that by definition, the empty set '$\emptyset$' is a subset of every set. Note also that every set is a subset of itself. When we want to exclude this possibility, we must have a different relation: a set A is a *proper subset* of a set B if A is a subset of B and, additionally, A does not equal B. We denote this relation by $A \subset B$.

The *union* of two sets A and B is the set whose members are members of A or of B (or of both). Formally, the union of A and B is denoted $A \cup B$ and is defined as $\{x \mid x \in A \textit{ or } x \in B\}$. Of course, if some element is a member of both A and B it is a member of the union.

Example 1.4 Set union

$$\begin{array}{ccccc}
\emptyset & \cup & \{a, e, o\} & = & \{a, e, o\} \\
\{a, e, o\} & \cup & \{i, e, u\} & = & \{a, e, i, o, u\} \\
\{x \mid x \textit{ is an English vowel}\} & \cup & \{a, e, i, o, u\} & = & \{x \mid x \textit{ is an English vowel}\} \\
\{x \mid x \textit{ is an English vowel}\} & \cup & \{x \mid x \textit{ is an English consonant}\} & = & \{x \mid x \textit{ is an English letter}\}
\end{array}$$

Note that for every set A, $A \cup \emptyset = A$ and $A \cup A = A$. Also, if $A \subseteq B$ then $A \cup B = B$.

© 2001 by Shuly Wintner

The *intersection* of two sets A and B is the set whose members are members of both A and B. Formally, the intersection of A and B is denoted $A \cap B$ and is defined as $\{x \mid x \in A \textit{ and } x \in B\}$.

Example 1.5 Set intersection

$$\begin{array}{ccccc}
\emptyset & \cap & \{a, e, o\} & = & \emptyset \\
\{a, e, o\} & \cap & \{i, e, u\} & = & \{e\} \\
\{x \mid x \textit{ is an English vowel}\} & \cap & \{a, e, i, o, u\} & = & \{a, e, i, o, u\} \\
\{x \mid x \textit{ is an English vowel}\} & \cap & \{x \mid x \textit{ is an English consonant}\} & = & \emptyset \\
\{x \mid x \textit{ is an English letter}\} & \cap & \{x \mid x \textit{ is an English vowel}\} & = & \{x \mid x \textit{ is an English vowel}\}
\end{array}$$

Here, notice that the intersection of any set with the empty set yields the empty set, and the intersection of any set A with itself yields A. If $A \subseteq B$ then $A \cap B = A$.

Another useful operation on sets is *set difference*: the difference between A and B, denoted $A \setminus B$, is defined as $\{x \mid x \in A \textit{ and } x \notin B\}$, where $x \notin B$ means x is not a member of B.

Example 1.6 Set difference

$$\begin{array}{ccccc}
\{a, e, o\} & \setminus & \emptyset & = & \{a, e, o\} \\
\{a, e, o\} & \setminus & \{i, e, u\} & = & \{a, o\} \\
\{x \mid x \textit{ is an English vowel}\} & \setminus & \{a, e, i, o, u\} & = & \emptyset \\
\{x \mid x \textit{ is an English letter}\} & \setminus & \{x \mid x \textit{ is an English vowel}\} & = & \{x \mid x \textit{ is an English consonant}\}
\end{array}$$

Finally, we define the *complement* of a set A, denoted $\overline{A}$, as all those elements not in A. In order to define this notion formally, we must set some universe of objects that might be referred to: it isn't very useful to assume the universe of "everything". For example, when reasoning about languages, the universe might be taken to be all possible utterances in some language. Then, if the set A is defined as $\{x \mid x \textit{ is an utterance of 12 words or less}\}$, the complement of A is $\overline{A} = \{y \mid y \textit{ is an utterance longer than 12 words}\}$.

1.2 Relations

Sets are unordered collections of elements. Sometimes, however, it is useful to have *ordered* collections of elements. We use angular brackets '$\langle\ \rangle$' to denote ordered *sequences*. For example, the sequence of characters in the English alphabet is

$$\langle a, b, c, d, e, f, g, h, i, j, k, l, m, n, o, p, q, r, s, t, u, v, w, x, y, z \rangle$$

The *length* of a sequence is the number of its elements; multiple occurrences of the same element are counted more than once. Since sequences are ordered, alternating the order of their elements changes the sequence. For example, $\langle a, e, i, o, u \rangle \neq \langle u, i, e, o, a \rangle$. Also, repetition of an element changes the sequence: $\langle a, e, i, o, u \rangle \neq \langle a, a, e, i, o, o, o, u \rangle$. Unlike sets, the concept of identity for sequences requires both identity of elements and identity of their position in the sequence. This implies that two sequences can only be equal if they are of the same length.

A special case of a sequence is an *ordered pair*: a sequence of length 2. If A and B are sets, consider the set obtained by collecting all the pairs in which the first element is a member of A and the second –

© 2001 by Shuly Wintner

a member of B. Formally, such a set is defined as $\{\langle x, y\rangle \mid x \in A \textit{ and } y \in B\}$. This set is called the *Cartesian product* of A and B and is denoted $A \times B$.

Example 1.7 Cartesian product

Let A be the set of all the vowels in some language and B the set of all consonants. For the sake of simplicity, take A to be $\{a, e, i, o, u\}$ and B to be $\{b, d, f, k, l, m, n, p, s, t\}$. The Cartesian product $B \times A$ is the set of all possible consonant–vowel pairs: $\{\langle b, a\rangle, \langle d, a\rangle, \langle d, i\rangle, \langle k, o\rangle, \langle p, o\rangle, \langle t, e\rangle, \langle t, u\rangle, \ldots\}$, etc. Notice that the Cartesian product $A \times B$ is different: it is the set of all vowel–consonant pairs, which is a completely different entity (albeit with the same number of elements). The Cartesian product $B \times B$ is the set of all possible consonant–consonant pairs, whereas $A \times A$ is the set of all possible diphthongs.

The Cartesian product is instrumental in the definition of a very powerful concept, that of *relations*. When A and B are sets, a relation from A to B is a subset of the Cartesian product of A and B. Formally, R is a relation from A to B if and only if $R \subseteq A \times B$. If R is a relation from A to B, it is a set of pairs; when a pair $\langle x, y\rangle$ is in the set (i.e., when $\langle x, y\rangle \in R$), we write $R(x, y)$, or sometimes xRy, and we say that R *holds* for $\langle x, y\rangle$.

Example 1.8 Relation

Let A be the set of all articles in German and B the set of all German nouns. The Cartesian product $A \times B$ is the set of all article–noun pairs. Any subset of this set of pairs is a relation from A to B. In particular, the set $R = \{\langle x, y\rangle \mid x \in A \textit{ and } y \in B \textit{ and } x \textit{ and } y \textit{ agree on number, gender and case}\}$ is a relation. Informally, R holds for all pairs of article–noun which form a grammatical noun phrase in German: such a pair is in the relation if and only if the article and the noun agree.

Relations are a special case of sets. In particular, the concept of a "universal domain" for relations from A to B is well defined: it is the entire Cartesian product of A and B. It is therefore possible to define the *complement* of a relation: if R is a relation from A to B, its complement is denoted $\overline{R}$ and is defined as $(A \times B) \setminus R$, that is, as all those pairs in the Cartesian product that are not members in R.

Another useful notion is that of the *inverse* of a relation. If R is a relation from A to B, then its inverse, denoted R^{-1}, is defined as $\{\langle x, y\rangle \mid \langle y, x\rangle \in R\}$. That is, the inverse or R consists of all the pairs of R, each in a reverse order. R^{-1} is a relation from B to A.

A very important kind of relation is a *function*. A relation R from A to B is said to be *functional* if and only if it pairs each member $x \in A$ with exactly one member of B.

Example 1.9 Function

Let $A = \{\textit{apple, banana, strawberry, grapefruit}\}$ and $B = \{\textit{green, yellow, red}\}$. The Cartesian product of A and B is the set which pairs each fruit in A with each color in B. The relation which relates each fruit with its "true" color is $R = \{\langle \textit{apple, green}\rangle, \langle \textit{banana, yellow}\rangle, \langle \textit{strawberry, red}\rangle, \langle \textit{grapefruit, yellow}\rangle\}$.
The inverse of R is $R^{-1} = \{\langle \textit{green, apple}\rangle, \langle \textit{yellow, banana}\rangle, \langle \textit{red, strawberry}\rangle, \langle \textit{yellow, grapefruit}\rangle\}$. Notice that R^{-1} no longer relates fruit to their colors; rather, it relates colors to the fruits that bear them.
The relation R is a function. Informally, this is because each fruit has exactly one color. The relation R^{-1} is not a function, since some colors (e.g., yellow) are the colors of many fruits.

Relations are defined over two sets. A special case of relations that is of special interest is relations in which the two sets are identical. If R is a relation from a set A to itself, we say that R is defined *in* A. We discuss some properties of such relations below.

A relation R in A is *reflexive* if and only if for every $x \in A$, xRx, that is, if every element in A is related to itself. A relation R in A is *symmetric* if whenever xRy also yRx; that is, whenever two elements

© 2001 by Shuly Wintner

are members of R they are also members of R^{-1}; it is *asymmetric* if whenever xRy, it implies that yRx does not hold. A relation R in A is *anti-symmetric* if whenever xRy and yRx we can deduce that $x = y$. Finally, a relation R in A is *transitive* if whenever xRy and yRz, also xRz.

Example 1.10 Properties of relations

Let A be a set of human beings; say, the set of residents of Finland. Let R_1 be a relation in A, defined by xR_1y if and only if x and y are relatives. In other terms, R_1 is the set $\{\langle x, y\rangle \mid x \textit{ is a relative of } y\}$. The relation R_1 is reflexive, because every person is a relative of himself; it is symmetric, because if I am your relative then you are my relative; it is therefore neither asymmetric nor anti-symmetric; and it is transitive, since if x is a relative of y and y is a relative of z, then x is a relative of z (through y).
Let R_2 be the relation defined by xR_2y if and only if x is a parent of y. R_2 is not reflexive, since it is not the case that every person is his/her own parent; it is not symmetric, since it is not the case that my child is my parent; in fact, it is asymmetric since whenever x is a parent of y, y is *not* a parent of x; and it is therefore vacuously anti-symmetric. R_2 is not transitive: if x is a parent of y and y is a parent of z, it does not follow that x is a parent of z. However, the relation R_3 defined as xR_3y if and only if x is an ancestor of y *is* transitive.

So far we have only looked at *binary* relations, that is, relations over *two* sets. However, it is possible to extend this notion to more than two sets: for example, a ternary relation, or a relation over three sets, A, B and C, is defined as a subset of the Cartesian product of A, B and C, that is, it is a collection of *triples*, where the first element is a member of A, the second – of B and the third – of C. In the same way we can define n-ary relations for any natural number n.

1.3 Strings

Formal languages are defined with respect to a given *alphabet*. The alphabet is simply a finite set of symbols, each of which is called a *letter*. This notation does not mean, however, that elements of the alphabet must be "ordinary" letters; they can be any symbols, such as numbers, or digits, or words. It is customary to use 'Σ' to denote the alphabet. A finite sequence of letters is called a *string*. For simplicity, we usually forsake the traditional sequence notation in favor of a more straight-forward representation of strings.

Example 1.11 Strings

Let $\Sigma = \{0, 1\}$ be an alphabet. Then all binary numbers are strings over Σ. Instead of $\langle 0, 1, 1, 0, 1\rangle$ we usually write 01101.
If $\Sigma = \{a, b, c, d, \ldots, y, z\}$ is an alphabet then *cat*, *incredulous* and *supercalifragilisticexpialidocious* are strings, as are *tac*, *qqq* and *kjshdflkwjehr*.

The *length* of a string w is the number of letters in the sequence, and is denoted $|w|$. A very special string is the unique string of length 0. It is called the *empty string* and is usually denoted ϵ (but sometimes λ).

If w_1 and w_2 are two strings over the same alphabet Σ, we can create a new word by writing w_2 following w_1. This operation is called *concatenation*, and is formally defined as follows: let $w_1 = \langle x_1, \ldots, x_n\rangle$ and $w_2 = \langle y_1, \ldots, y_m\rangle$. The concatenation of w_1 and w_2, denoted $w_1 \cdot w_2$, is the string $\langle x_1, \ldots, x_n, y_1, \ldots, y_m\rangle$. Note that the length of $w_1 \cdot w_2$ is the sum of the lengths of w_1 and w_2: $|w_1 \cdot w_2| = |w_1| + |w_2|$. When it is clear from the context, we sometimes omit the '$\cdot$' symbol when depicting concatenation.

© 2001 by Shuly Wintner

Example 1.12 Concatenation

Let $\Sigma = \{a, b, c, d, \ldots, y, z\}$ be an alphabet. Then *master* $\cdot$ *mind* = *mastermind*, *mind* $\cdot$ *master* = *mindmaster* and *master* $\cdot$ *master* = *mastermaster*. Similarly, *learn* $\cdot$ *s* = *learns*, *learn* $\cdot$ *ed* = *learned* and *learn* $\cdot$ *ing* = *learning*.

Notice that when the empty string ϵ is concatenated with any string w, the resulting string is w. Formally, for every string w, $w \cdot \epsilon = \epsilon \cdot w = w$.

We define an *exponent* operator over strings in the following way: for every string w, w^0 (read: w raised to the power of zero) is defined as ϵ. Then, for $n > 0$, w^n is defined as $w^{n-1} \cdot w$. Informally, w^n is obtained by concatenating w with itself n times. In particular, $w^1 = w$.

Example 1.13 Exponent

If $w = go$, then $w^0 = \epsilon$, $w^1 = w = go$, $w^2 = w^1 \cdot w = w \cdot w = gogo$, $w^3 = gogogo$ and so on.

A few other notions that will be useful in the sequel: the *reversal* of a string w is denoted w^R and is obtained by writing w in the reverse order. Thus, if $w = \langle x_1, x_2, \ldots, x_n \rangle$, $w^R = \langle x_n, x_{n-1}, \ldots, x_1 \rangle$.

Example 1.14 Reversal

Let $\Sigma = \{a, b, c, d, \ldots, y, z\}$ be an alphabet. If w is the string *saw*, then w^R is the string *was*. If $w = madam$ then $w^R = madam = w$. In this case we say that w is a *palindrome*.

Given a string w, a *substring* of w is a sequence formed by taking contiguous symbols of w in the order in which they occur in w. If $w = \langle x_1, \ldots, x_n \rangle$ then for any i, j such that $1 \leq i \leq j \leq n$, $\langle x_i, \ldots x_j \rangle$ is a substring of w. Another way to define substrings is by saying that w_c is a substring of w if and only if there exist (possibly empty) strings w_l and w_r such that $w = w_l \cdot w_c \cdot w_r$. Two special cases of substrings are *prefix* and *suffix*: if $w = w_l \cdot w_c \cdot w_r$ then w_l is a prefix of w and w_r is a suffix of w. Note that every prefix and every suffix is a substring, but not every substring is a prefix or a suffix.

Example 1.15 Substrings

Let $\Sigma = \{a, b, c, d, \ldots, y, z\}$ be an alphabet and w = *indistinguishable* a string over Σ. Then ϵ, *in, indis, indistinguish* and *indistinguishable* are prefixes of w, while ϵ, *e, able, distinguishable* and *indistinguishable* are suffixes of w. Substrings that are neither prefixes nor suffixes include *distinguish, gui* and *is*.

1.4 Languages

Given an alphabet Σ, the set of all strings over Σ is denoted by Σ^* (the reason for this notation will become clear presently). Notice that no matter what the alphabet is, Σ^* is always infinite. Even for an impoverished alphabet that consists of a single letter only, say $\Sigma = \{a\}$, Σ^* contains the strings $a^0 = \epsilon$, $a^1 = a$, $a^2 = aa$, etc., and in general for every $k > 0$ the string a^k is in Σ^*. Since one cannot impose a bound on the length of a string in Σ^*, it is impossible to bound the number of its elements, and hence Σ^* is infinite.

A *formal language* over an alphabet Σ is simply a subset of Σ^*. Any subset, be it finite or infinite, is a language. Since Σ^* is always infinite, the number of its subsets is also infinite. In other words, given any alphabet Σ, even an extremely impoverished alphabet consisting of a single character, there are infinitely many formal languages over Σ! Some possible (formal) languages over a "natural" alphabet are depicted in the following example.

© 2001 by Shuly Wintner

Example 1.16 Languages

Let $\Sigma = \{a, b, c, \ldots, y, z\}$. Then Σ^* is the set of all strings over the Latin alphabet. Any subset of this set is a language. In particular, the following are formal languages:

- Σ^*;
- the set of strings consisting of consonants only;
- the set of strings consisting of vowels only;
- the set of strings each of which contains at least one vowel and at least one consonant;
- the set of palindromes: strings that read the same from right to left or from left to right;
- the set of strings whose length is less than 17 letters;
- the set of single-letter strings;
- the set $\{$*i, you, he, she, it, we, they*$\}$;
- the set of words occurring in Joyce's Ulysses;
- the empty set;

Note that the first five languages are infinite while the last five are finite.

We can now "lift" some of the string operations defined above to languages. If L is a language then the *reversal* of L, denoted L^R, is the language $\{w \mid w^R \in L\}$, that is, the set of reversed L-strings. *Concatenation* can also be lifted to languages: if L_1 and L_2 are languages, then $L_1 \cdot L_2$ is the language defined as $\{w_1 \cdot w_2 \mid w_1 \in L_1 \textit{ and } w_2 \in L_2\}$: the concatenation of two languages is the set of strings obtained by concatenating some word of the first language with some word of the second.

Example 1.17 Language operations

Let $L_1 = \{$*i, you, he, she, it, we, they*$\}$ and $L_2 = \{$*smile, sleep*$\}$. Then ${L_1}^R = \{$*i, uoy, eh, ehs, ti, ew, yeht*$\}$ and $L_1 \cdot L_2 = \{$*ismile, yousmile, hesmile, shesmile, itsmile, wesmile, theysmile, isleep, yousleep, hesleep, shesleep, itsleep, wesleep, theysleep*$\}$.

In the same way we can define the *exponent* of a language: if L is a language then L^0 is the language containing the empty string only, $\{\epsilon\}$. Then, for $i > 0$, $L^i = L \cdot L^{i-1}$, that is, L^i is obtained by concatenating L with itself i times.

Example 1.18 Language exponentiation

Let L be the set of words $\{$*bau, haus, hof, frau*$\}$. Then $L^0 = \{\epsilon\}$, $L^1 = L$ and $L^2 = \{$*baubau, bauhaus, bauhof, baufrau, hausbau, haushaus, haushof, hausfrau, hofbau, hofhaus, hofhof, hoffrau, fraubau, frauhaus, frauhof, fraufrau*$\}$.

Sometimes it is useful to consider not a particular number of concatenations, but "any" number. Informally, given a language L one might want to consider the language obtained by concatenating some word of L with some other word of L, then concatenating the result with yet another word, and so on and so forth. The language obtained by considering any number of concatenations of words from L is called the *Kleene closure* of L and is denoted L^*. Formally, it is defined as $\bigcup_{i=0}^{\infty} L^i$, which is a terse notation for the

© 2001 by Shuly Wintner

unioning of L^0 with L^1, then with L^2, L^3 and so on ad infinitum. When one wants to leave L^0 out, one writes $L^+ = \bigcup_{i=1}^{\infty} L^i$.

Example 1.19 Kleene closure

Let $L = \{$*dog, cat*$\}$. Observe that $L^0 = \{\epsilon\}$, $L^1 = \{$*dog, cat*$\}$, $L^2 = \{$*catcat, catdog, dogcat, dogdog*$\}$, etc. Thus L^* contains, among its infinite set of strings, the strings ϵ, *cat, dog, catcat, catdog, dogcat, dogdog, catcatcat, catdogcat, dogcatcat, dogdogcat*, etc.
As another example, consider the alphabet $\Sigma = \{a, b\}$ and the language $L = \{a, b\}$ defined over Σ. L^* is the set of all strings over a and b, which is exactly the definition of Σ^*. The notation for Σ^* should now become clear: it is simply a special case of L^*, where $L = \Sigma$.

Further reading

Most of the material presented in this chapter can be found in any introductory textbook on set theory of formal language theory. A very formal presentation of this material is given by Hopcroft and Ullman (1979, chapter 1). Just as rigorous, but with an eye to linguistic uses and applications, is the presentation of Partee, ter Meulen, and Wall (1990, chapters 1–3).

© 2001 by Shuly Wintner

Chapter 2

Regular languages

2.1 Regular expressions

A formal language is simply a set of strings, a subset of Σ^*, and we have seen a variety of languages in the previous chapter. As languages are sets, they can be specified using any of the specification methods for sets discussed in section 1.1. However, when languages are fairly regular, they can be characterized by more regular means. In other words, if the languages one is interested in are not arbitrarily complex, a formal and simple means of specifying them can be more useful than the general, rather informal predicate notation. For example, suppose Σ is an alphabet, say $\Sigma = \{a, b\}$, and suppose one wants to specify the language that contains all and only the strings which start with three a's and end in three b's. There is an infinite number of such strings, of course, and therefore an enumeration of the members of the language is not an option. In predicate notation, this language can be described as $\{w \mid w \in \Sigma^*$ and w starts with "aaa" and ends with "bbb"$\}$. However, this is not a formal notation, and when informal specification are used, one runs the risk of vague and ambiguous specifications. It is advantageous, then, to have a more formal means for specifying languages.

We now present such a formalism, called *regular expressions*. It is a meta-language: a language for specifying formal languages. Its "syntax" is formally defined and is relatively simple. Its "semantics" will be sets of strings in a formal language. The formalism defines regular expressions: formal expressions over some alphabet Σ, augmented by some special characters. It also defines a mapping, called *denotation*, from regular expressions to sets of strings over Σ. In other words, every well-formed regular expression denotes a set of strings, or a language. Regular expressions over an alphabet Σ are defined as follows:

- $\emptyset$ is a regular expression
- ϵ is a regular expression
- if $a \in \Sigma$ is a letter then a is a regular expression
- if r_1 and r_2 are regular expressions then so are $(r_1 + r_2)$ and $(r_1 \cdot r_2)$
- if r is a regular expression then so is $(r)^*$
- nothing else is a regular expression over Σ.

Example 2.1 Regular expressions

Let Σ be the alphabet $\{a, b, c, \ldots, y, z\}$. Some regular expressions over this alphabet are $\emptyset$, a, $((c \cdot a) \cdot t)$, $(((m \cdot e) \cdot (o)^*) \cdot w)$, $(a + (e + (i + (o + u))))$, $((a + (e + (i + (o + u)))))^*$, etc.

Now that the form of regular expressions is defined, we can define their meanings. For every regular expression r its denotation, $[\![r]\!]$, is a set of strings defined as follows:

- $[\![\emptyset]\!]$ is the empty set $\emptyset$
- $[\![\epsilon]\!]$ is the singleton set containing the empty string only: $\{\epsilon\}$
- if $a \in \Sigma$ is a letter then $[\![a]\!]$ is the singleton set containing a only: $\{a\}$
- if r_1 and r_2 are two regular expressions whose denotations are $[\![r_1]\!]$ and $[\![r_2]\!]$, respectively, then $[\![(r_1 + r_2)]\!]$ is $[\![r_1]\!] \cup [\![r_2]\!]$ and $[\![(r_1 \cdot r_2)]\!]$ is $[\![r_1]\!] \cdot [\![r_2]\!]$
- if r is a regular expression whose denotation is $[\![r]\!]$ then the denotation of $(r)^*$ is $[\![r]\!]^*$

Example 2.2 Regular expressions and their denotations

Following are the denotations of the regular expressions of the previous example:

$\emptyset$	$\emptyset$
ϵ	$\{\epsilon\}$
a	$\{a\}$
$((c \cdot a) \cdot t)$	$\{c \cdot a \cdot t\}$
$(((m \cdot e) \cdot (o)^*) \cdot w)$	$\{$*mew, meow, meoow, meooow, meoooow,* $\ldots\}$
$(a + (e + (i + (o + u))))$	$\{a, e, i, o, u\}$
$((a + (e + (i + (o + u)))))^*$	*the set containing all strings of 0 or more vowels*

Regular expressions are useful because they enable us to specify complex languages in a formal, concise way. Of course, finite languages can still be specified by enumerating their members; but infinite languages are much easier to specify with a regular expression, as the last instance of the above example shows.

For simplicity, we will omit the parentheses around regular expressions when no confusion can be caused. Thus, the expression $((a + (e + (i + (o + u)))))^*$ will be written as $(a + e + i + o + u)^*$. Also, if $\Sigma = \{a_1, a_2, \ldots, a_n\}$, we will use Σ as a shorthand notation for $a_1 + a_2 + \cdots + a_n$. As was the case with string concatenation and language concatenation, we sometimes omit the '$\cdot$' operator in regular expressions, so that the expression $c \cdot a \cdot t$ can be written *cat*.

Example 2.3 Regular expressions

To exemplify the power and expressiveness of regular expressions, we present here a few expressions denoting rather complex languages.

Given the alphabet of all English letters, $\Sigma = \{a, b, c, \ldots, y, z\}$, the language Σ^* is denoted by the regular expression Σ^* (recall our convention of using Σ as a shorthand notation). The set of all strings which contain a vowel is denoted by $\Sigma^* \cdot (a + e + i + o + u) \cdot \Sigma^*$. The set of all strings that begin in "*un*" is denoted by $(un)\Sigma^*$. The set of strings that end in either "*tion*" or "*sion*" is denoted by $\Sigma^* \cdot (s + t) \cdot (ion)$. Note that all these languages are infinite.

What languages can be expressed as the denotations of regular expressions? This is an important question which we will discuss in detail in section 4.2. For the time being, suffice it to say that not *every* language can be thus specified. The set of languages which can be expressed as the denotation of regular expressions is called *regular languages*. It is a mathematical fact that some languages, subsets of Σ^*, are not regular. We will encounter such languages in the sequel.

© 2001 by Shuly Wintner

2.2 Properties of regular languages

We have defined the class of regular languages above as the class of languages that can be specified by regular expressions. It is important to note that this is a set of languages, that is, a set of sets of strings. In this section we discuss some properties of this class of languages. This will be related to a discussion of the expressiveness of this class, which we defer to section 4.2.

When classes of languages are discussed, some of the interesting properties to be investigated are *closures* with respect to certain operators. Recall from section 1.4 that several operators, such as concatenation, union, Kleene-closure etc., are defined over languages. Given a particular operation, say union, it is interesting to know whether the class of regular languages is *closed under* this operation. When we say that regular languages are closed under union, for example, we mean that whenever we take the union of two regular languages, the result – which is a language – is guaranteed to be a regular language. More generally, a class of languages $\mathcal{L}$ is said to be closed under some operation '$\bullet$' if and only if whenever two languages L_1, L_2 are in the class ($L_1, L_2 \in \mathcal{L}$), also the result of performing the operation on the two languages is in this class: $L_1 \bullet L_2 \in \mathcal{L}$.

Closure properties have a theoretical interest in and by themselves, but they are especially important when one is interested in processing languages. For example, if we have an efficient computational implementation for regular languages (say, one which could tell us whether a given string indeed belongs to a given language), and we know that certain languages are regular, then we can freely use the operators that the regular languages are closed under, and still preserve computational efficiency in processing. At a more concrete level, suppose we have a part-of-speech tagger: a program that determines the part of speech of each word in a running text, when the words are members of some regular language. Suppose we now want to extend the language so that the coverage of the tagger is improved. We can simply test our tagger on the language extension: those words that are not in the original language. Assuming that this is also a regular language, and assuming that the regular languages are closed under union, we can simply take the union of the two languages: the original one and its extension, and the tagger should be able to cope efficiently with this new language, as it is guaranteed to be regular.

What operations, then, are the regular languages closed under? It should be fairly easy to see that they are closed under union, concatenation and Kleene-closure. Since any regular language, by definition, is the denotation of some regular expression, whenever we have a regular language we can refer to some regular expression (perhaps not unique) of which it is the denotation. Thus, when given two regular languages, L_1 and L_2, there always exist two regular expressions, r_1 and r_2, such that $[\![r_1]\!] = L_1$ and $[\![r_2]\!] = L_2$. It is therefore possible to form new regular expressions based on r_1 and r_2, such as $r_1 \cdot r_2$, $r_1 + r_2$ and r_1^*. Now, by the definition of regular expressions and their denotations, it follows that the denotation of $r_1 \cdot r_2$ is $L_1 \cdot L_2$: $[\![r_1 \cdot r_2]\!] = L_1 \cdot L_2$. Since $r_1 \cdot r_2$ is a regular expression, its denotation is a regular language, and hence $L_1 \cdot L_2$ is a regular language. Since L_1 and L_2 are arbitrary (regular) languages, the logical process outlined above is valid for any two regular languages: hence the regular languages are closed under concatenation. In exactly the same way we can prove that the class of regular languages is closed under union and Kleene-closure.

Of course, the fact that the regular languages are closed under some operations does not imply that they are closed under any operation. However, one of the reasons for the attractiveness of regular languages is that they are known to be closed under a wealth of useful operations: intersection, complementation, exponentiation, substitution, homomorphism, etc. These properties come in very handy both in practical applications that use regular languages and in mathematical proofs that concern them. For example, there exist several extended regular expression formalisms that allow the user to express regular languages using not only the three operators that define regular expressions, but also a wealth of other operators (that the class of regular languages is closed under). It is worth noting that such "good behavior" is not exhibited by more complex classes of languages, such as the context-free languages that we will meet in chapter 3. See section 3.7 for more details.

© 2001 by Shuly Wintner

2.3 Finite-state automata

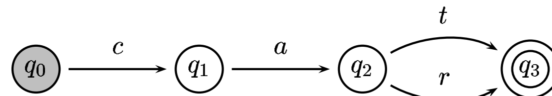
In the previous chapter we presented formal languages as abstract entities: sets of strings of letters taken from some predefined alphabet. In section 2.1 we presented a meta-language, regular expressions, which is a formalism for specifying languages. We also defined the set of languages that can be denoted by regular expressions, namely the set of regular languages. This section is dedicated to a different take on languages: we present languages as entities generated by a *computation*. This is a very common situation in formal language theory: many language families (such as the class of regular languages) are associated with computing machinery that generates them. The dual view of languages (as the denotation of some specifying formalism and as the output of a computational process) is central in formal language theory.

The computational device we define in this section is a very simple one. The easy way to envision it would be to think of a finite set of *states*, connected by a finite number of *transitions*. Each of the transitions is labeled by a letter, taken from some finite alphabet, and we will continue to use Σ for the alphabet. A computation starts at a designated state, the *start* state or *initial* state, and it moves from one state to another along the labeled transitions. As it moves, it prints the letter which labels the transition. Thus, during a computation, a string of letters is printed out. Now some of the states of the machine are designated *final* states, or *accepting* states. Whenever the computation reaches a final state, the string that was printed so far is said to be *accepted* by the machine. Since each computation defines a string, the set of all possible computations defines a set of strings, or in other words, a language. We say that this language is *accepted* or *generated* by the machine.

The computational devices we define here essentially contain a finite number of states. Therefore, they are called *finite-state* automata. In order to visualize finite-state automata we will usually depict them graphically. The states will be depicted as circles; the initial state will be shaded, and will usually be the leftmost (or, sometimes, topmost) state in the picture; the final states will be depicted by two concentric circles. The transitions will be depicted as arcs connecting the circles, with the labels placed next to them. The alphabet will usually not be explicitly given: we will usually assume that the alphabet is simply the collection of all the letters that label the arcs in an automaton.

Example 2.4 Finite-state automaton

The following automaton has four states: q_0, q_1, q_2 and q_3. It has four arcs: one, labeled c, connects q_0 with q_1; the second, labeled a, connects q_1 with q_2; and the other two, labeled t and r, connect q_2 with q_3. The initial state is q_0, which is shaded and depicted as the leftmost state, and the only final state is q_3, as is evident from the double circle.



The above presentation of finite-state automata was rather intuitive and informal. Formally, a finite-state automaton is a five-tuple, or a sequence of five elements. The first element is a finite set of states; it is customary to use Q for this set. The next element is an alphabet, or a finite set of letters, and we use Σ as usual for this set. Then, the third element is the start state of the automaton, which must of course be an element in Q; we use q_0 for this state. The fourth element, usually δ, is a relation from states and alphabet symbols to states. Finally, the last element, F, is a set of final states, so that $F \subseteq Q$. Thus, an automaton A is written as $\langle Q, \Sigma, q_0, \delta, F \rangle$.

Example 2.5 Finite-state automaton

The finite-state automaton of example 2.4 can be formally written as $A = \langle Q, \Sigma, q_0, \delta, F \rangle$, where $Q = \{q_0, q_1, q_2, q_3\}$, $\Sigma = \{c, a, t, r\}$, $F = \{q_3\}$ and $\delta = \{\langle q_0, c, q_1 \rangle, \langle q_1, a, q_2 \rangle, \langle q_2, t, q_3 \rangle, \langle q_2, r, q_3 \rangle\}$.

© 2001 by Shuly Wintner

Now that automata are formally defined, we can discuss the way by which they generate, or accept, languages. Informally, when looking at a graphical depiction of an automaton, one can construct a "path" by starting in the initial state, moving along the transition arcs (in the direction indicated by the arrows) and reaching one of the final states. During such a journey, the sequence of letters that label the arcs traversed, in the order in which they were traversed, constitutes a string, and this string is said to be accepted by the automaton.

To capture this notion formally, we first extend the transition relation from single arcs to "paths". Formally, we define the reflexive transitive extension of the transition relation δ as a new relation, $\hat{\delta}$, from states and strings to states. The extended relation is defined as follows: first, for every state $q \in Q$, $\hat{\delta}(q, \epsilon, q)$ holds. In other words, the empty string can lead from each state to itself, but nowhere else. In addition, if a certain string w labels the path from the initial state to some state q, and there is an outgoing transition from q to q' labeled a, then the string $w \cdot a$ labels a path from the initial state to q'. Formally, for every string $w \in \Sigma^*$ and letter $a \in \Sigma$, if $\hat{\delta}(q, w, q')$ and $\delta(q', a, q'')$ then $\hat{\delta}(q, w \cdot a, q'')$.

Example 2.6 Paths

For the finite-state automaton of example 2.4, $\hat{\delta}$ is the following set of triples:

$$\langle q_0, \epsilon, q_0\rangle, \langle q_1, \epsilon, q_1\rangle, \langle q_2, \epsilon, q_2\rangle, \langle q_3, \epsilon, q_3\rangle, \langle q_0, c, q_1\rangle, \langle q_1, a, q_2\rangle, \langle q_2, t, q_3\rangle, \langle q_2, r, q_3\rangle,$$
$$\langle q_0, ca, q_2\rangle, \langle q_1, at, q_3\rangle, \langle q_1, ar, q_3\rangle, \langle q_0, cat, q_3\rangle, \langle q_0, car, q_3\rangle$$

Note that the middle element of each triple is a string, rather than a letter as was the case with δ.

A string w is accepted by an automaton if and only if there is a path labeled w which leads from the initial state to any of the final states. Formally, we write $w \in L(A)$ and say that w is accepted by the automaton $A = \langle Q, q_0, \Sigma, \delta, F\rangle$ if and only if there exists a state $q_f \in F$ such that $\hat{\delta}(q_0, w) = q_f$. The *language accepted by a finite-state automaton* is the set of all strings it accepts.

Example 2.7 Language accepted by an automaton

The automaton of example 2.4 accepts exactly two strings: *cat*, along the path $q_0 - q_1 - q_2 - q_3$, and *car*, along the same path (albeit with a different final transition). Therefore, its language is $\{$*cat, car*$\}$.

Automata are computational devices for computing languages, and so an automaton can be viewed as an alternative definition for a language. In example 2.8 (page 14) we present a number of automata generating some languages we encountered in this chapter.

We introduce the convention of writing '?' for "any character of Σ" in example 2.8. Another convention we will use in subsequent examples of automata would be to label a single transition by more than one symbol, instead of drawing several transitions between the two nodes, each labeled by one of the symbols.

We now slightly amend the definition of finite-state automata to include what is called ϵ-moves. By our original definition, the transition relation δ is a relation from states and alphabet symbols to states. We now extend δ: the transition relation is now defined over states, but rather than alphabet symbols, its second coordinate is now $\Sigma \cup \{\epsilon\}$, that is, any arc in an automaton can be labeled either by some alphabet symbol or by the special symbol ϵ, which as usual denotes the empty word. The implication is that while traversing the automaton, one can transit from one state to another without printing out any symbol, or in other words, that a word w can be accepted by the automaton by traversing more arcs than the number of characters in w. It is possible to show that for every finite-state automaton with ϵ-transitions there exists an equivalent finite-state automaton (accepting the same language) without ϵ-moves.

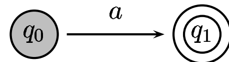
© 2001 by Shuly Wintner

Example 2.8 Some finite-state automata

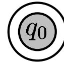
We start with an automaton which accepts no string at all, that is, which generates the empty language. This automaton has a single state, which by definition has to be the initial state, so $Q = \{q_0\}$. Since we don't want the automaton to accept any string, no final states are necessary, and hence $F = \emptyset$. For the same reason, no transitions are required, and thus $\delta = \emptyset$ as well. Graphically, this automaton is depicted as follows:

Since there are no final states, for no string w does there exist a state $q_f \in F$ such that $\hat{\delta}(q_0, w, q_f)$ is defined, and hence the language accepted by the automaton is empty.

Now consider the language which consists of a single string, a. An automaton which accepts this language must have at least one accepting state, and indeed one such state suffices. There must be a transition labeled a leading from the initial state to this single final state. A finite-state A whose language $L(A)$ is $\{a\}$ is:

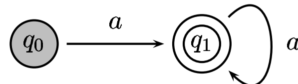


What if we wanted an automaton to accept a language of one string only, but we wanted the accepted string to be the empty string? In other words, what would an automaton A look like when $L(A) = \{\epsilon\}$? Again, a single final state is required, but since we want the machine to accept the empty string, the final state must coincide with the initial state. No transitions are necessary, as we don't want the automaton to accept any other string. Here it is:

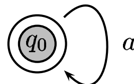


Note the difference from the first automaton in this example.

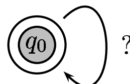
Next, consider an automaton the language of which consists of the set of strings of one or more 'a's: $\{$*a, aa, aaa, aaaa*$, \ldots\}$. We need to provide paths from the initial state to a final state labeled by a, by *aa, aaa*, etc., and the number of paths must be infinite. But since finite-state automata have a finite number of states, an exhaustive enumeration of these states is impossible. Instead, one could use a finite number of states in an infinite number of paths: this can be achieved using *loops*, or *circular* paths. The following automaton has only two states and only two transitions, but the language it generates is infinite. This is because the transition that leads from q_1 to itself can be traversed an infinite number of times. Since q_1 is a final state, it can be reached in infinitely many different ways: by walking the path a, or by walking the path *aa*, or *aaa*, and so on, ad infinitum. The language it accepts is therefore the required one.



Similarly, an automaton for accepting the language of zero or more occurrences of a would be:



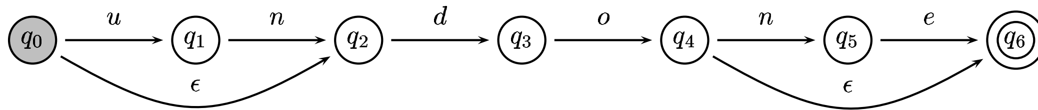
What will an automaton accepting Σ^* look like? We can use a convention by which the wildcard symbol '?' stands for any character of Σ, and depict the automaton as follows:



© 2001 by Shuly Wintner

Example 2.9 Automata with ϵ-moves

The language accepted by the following automaton is $\{$*do, undo, done, undone*$\}$:

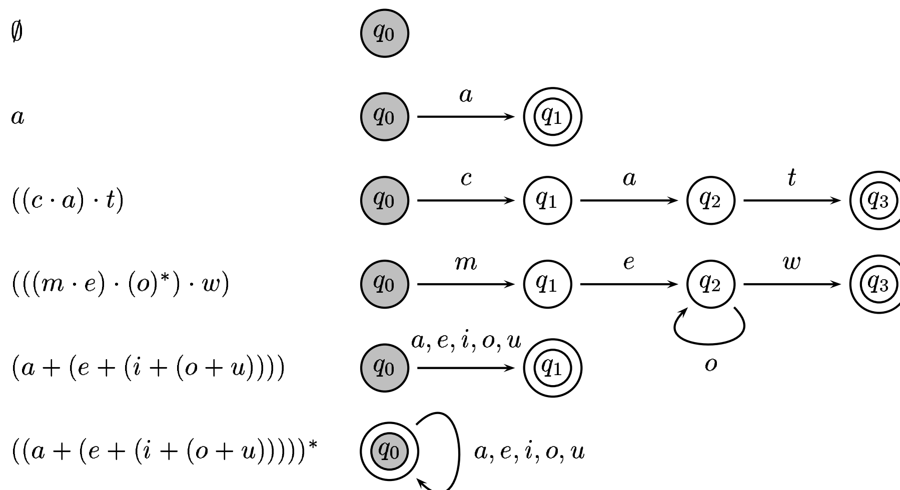


Note that in order for the automaton to accept, say, *undo*, an accepting path must traverse the ϵ-arc that connects q_4 with q_6. While traversing this arc, no symbol is read (or generated): the ϵ is taken to denote the empty word, as usual.

Finite-state automata, just like regular expressions, are devices for defining formal languages. As was the case with regular expressions, it is interesting to investigate the expressiveness of finite-state automata. Interestingly, it is a mathematical fact that the class of languages which can be generated by some finite-state automaton is exactly the class of regular languages. In other words, any language that is the denotation of some regular expression can be generated by some finite-state automaton, and any language that can be generated by some finite-state automaton is the denotation of some regular expression. Furthermore, there are simple and efficient algorithms for "translating" a regular expression to an equivalent automaton and vice versa.

Example 2.10 Equivalence of finite-state automata and regular expressions

For each of the regular expressions of example 2.2 we depict an equivalent automaton below:



2.4 Minimization and determinization

The finite-state automata presented above are non-deterministic. By this we mean that when the computation reaches a certain state, the next state is not uniquely determined by the next alphabet symbol to be printed. There might very well be more than one state that can be reached by a transition that is labeled by some symbol. This is because we defined automata using a transition relation, δ, which is not required to be functional. For some state q and alphabet symbol a, δ might include the two pairs $\langle q, a, q_1 \rangle$ and $\langle q, a, q_2 \rangle$ with $q_1 \neq q_2$. Furthermore, when we extended δ to allow ϵ-transitions we added yet another dimension

© 2001 by Shuly Wintner

of non-determinism: when the machine is in a certain state q and an ϵ-arc leaves q, the computation must "guess" whether to traverse this arc.

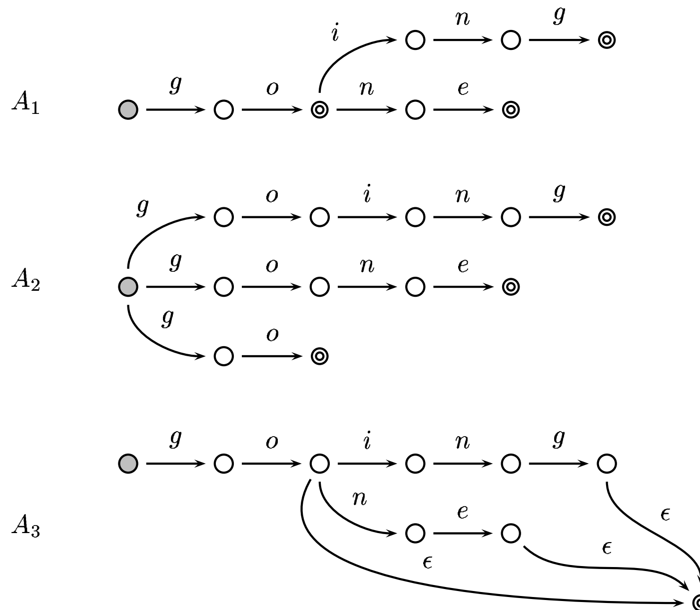
Much of the appeal of finite-state automata lies in their efficiency; and their efficiency is in great part due to the fact that, given some finite-state automaton A and a string w, it is possible to determine whether or not $w \in L(A)$ by "walking" the path labeled w, starting with the initial state of A, and checking whether the walk leads to a final state. Such a walk takes time that is proportional to the length of w. But when automata are non-deterministic, an element of guessing is introduced, which might very well impair the efficiency: no longer is there a single walk along a single path labeled w, and some control mechanism must be introduced to check that all possible paths are taken.

The good news is that non-determinism works for us: it is sometimes much easier to construct a non-deterministic automaton for some language. And we can rely on two very important results: every non-deterministic finite-state automaton is equivalent to some deterministic one; and every finite-state automaton is equivalent to one that has a minimum number of nodes, and the minimal automaton is unique. We now explain these two results.

First, it is important to clarify what is meant by "equivalent". We say that two finite-state automata are equivalent if and only if they accept the same language: every string that is accepted by one of the automata is accepted by the other. Note that two equivalent automata can be very different from each other, they can have different numbers of states, different accepting states etc. All that counts is the language that they generate. In the following example and in future ones, we suppress the identities of the nodes when depicting automata.

Example 2.11 Equivalent automata

The following three finite-state automata are equivalent: they all accept the set {*go, gone, going*}.



Note that A_1 is deterministic: for any state and alphabet symbol there is at most one possible transition. A_2 is not deterministic: the initial state has three outgoing arcs all labeled by g. The third automaton, A_3, has ϵ-arcs and hence is not deterministic. While A_2 might be the most readable, A_1 is the most compact as it has the fewest nodes.

As we said above, given a non-deterministic finite-state automaton A it is always possible to construct

© 2001 by Shuly Wintner

an equivalent deterministic automaton, one whose next state is fully determined by the current state and the alphabet symbol, and which contains no ϵ-moves. Sometimes this construction yields an automaton with more states than the original, non-deterministic one. However, this automaton can then be minimized such that it is guaranteed that no deterministic finite-state automaton generating the same language is smaller. Thus, it is always possible to determinize and then minimize a given automaton without affecting the language it generates. From an efficiency point of view, then, we can always deal with deterministic automata. For convenience, we can go on using non-determinism and ϵ-arcs in our automata, as these can always be removed.

2.5 Operations on finite-state automata

We know from the previous section that finite-state automata are equivalent to regular expressions, or in other words, that any regular language can be generated by some automaton. We also know from section 2.2 that the regular languages are closed under several operations, including union, concatenation and Kleene-closure. So, for example, if L_1 and L_2 are two regular languages, there exist automata A_1 and A_2 which accept them, respectively. Since we know that $L_1 \cup L_2$ is also a regular language, there must be an automaton which accepts it as well. The question is, can this automaton be constructed using the automata A_1 and A_2? In this section we show how simple operations on finite-state automata can correspond to some operators on languages.

We start with concatenation. Suppose that A_1 is a finite-state automaton such that $L(A_1) = L_1$, and similarly that A_2 is an automaton such that $L(A_2) = L_2$. We describe an automaton A such that $L(A) = L_1 \cdot L_2$. A word w is in $L_1 \cdot L_2$ if and only if it can be broken into two parts, w_1 and w_2, such that $w = w_1 \cdot w_2$, and $w_1 \in L_1, w_2 \in L_2$. In terms of automata, this means that there is an accepting path for w_1 in A_1 and an accepting path for w_2 in A_2; so if we allow an ϵ-transition from all the final states of A_1 to the initial state of A_2, we will have accepting paths for words of $L_1 \cdot L_2$. The finite-state automaton A is constructed by combining A_1 and A_2 in the following way: its set of states, Q, is the union of Q_1 and Q_2; its alphabet is the union of the two alphabets; its initial state is the initial state of A_1; its final states are the final states of A_2; and its transition relation is obtained by adding to $\delta_1 \cup \delta_2$ the set of ϵ-moves described above: $\{\langle q_f, \epsilon, q_{02}\rangle \mid q_f \in F_1\}$ where q_{02} is the initial state of A_2. This construction is graphically depicted in example 2.12 (page 18).

In a very similar way, an automaton A can be constructed whose languages is $L_1 \cup L_2$ by combining A_1 and A_2. Here, one should notice that for a word to be accepted by A it must either be accepted by A_1 or by A_2 (or by both). The combined automaton will have an accepting path for every accepting path in A_1 and in A_2. The idea is to add a new initial state to A, from which two ϵ-arcs will lead to the initial states of A_1 and A_2. The states of A will be the union of the states of A_1 and A_2, plus the new initial state. The transition relation will be the union of δ_1 with δ_2, plus the new ϵ-arcs. The final states will be the union of F_1 and F_2. This construction is depicted graphically in example 2.13 (page 18).

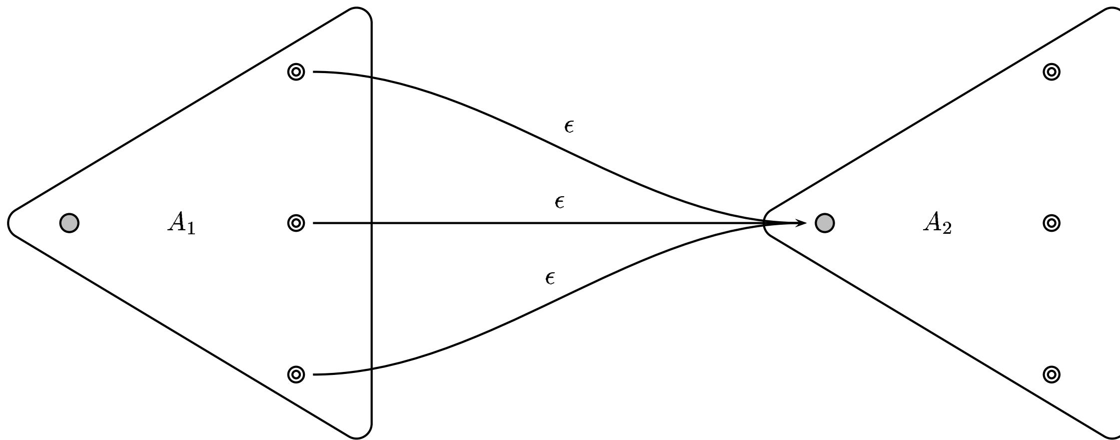
An extension of the same technique to construct the Kleene-closure of an automaton is rather straightforward. However, all these results are not surprising, as we have already seen in section 2.2 that the regular languages are closed under these operations. Thinking of languages in terms of the automata that accept them comes in handy when one wants to show that the regular languages are closed under other operations, where the regular expression notation is not very suggestive of how to approach the problem. Consider the operation of *complementation*: if L is a regular language over an alphabet Σ, we say that the complement of L is the set of all the words (in Σ^*) that are not in L, and write $\overline{L}$ for this set. Formally, $\overline{L} = \Sigma^* \setminus L$. Given a regular expression r, it is not clear what regular expression r' is such that $[\![r']\!] = \overline{[\![r]\!]}$. However, with automata this becomes much easier.

Assume that a finite-state automaton A is such that $L(A) = L$. Assume also that A is deterministic in the sense that its behavior is defined for every symbol in Σ, at every state. If this is not the case, simply determinize it while preserving its language. To construct an automaton for the complemented language,

© 2001 by Shuly Wintner

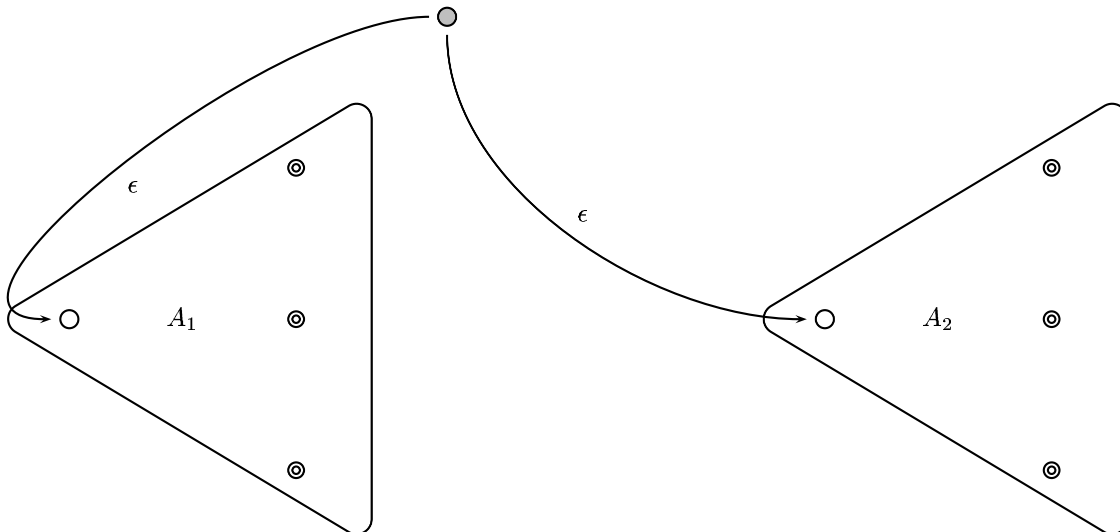
Example 2.12 Concatenation of finite-state automata

Suppose A_1 and A_2 are schematically depicted as the two triangles below. Then the combined automaton is obtained by adding the ϵ-arcs as shown below:



Example 2.13 Union of finite-state automata

An automaton A whose language is $L(A_1) \cup L(A_2)$:



all one has to do is change all final states to non-final, and all non-final states to final. In other words, if $A = \langle Q, \Sigma, q_0, \delta, F \rangle$, then $\overline{A} = \langle Q, \Sigma, q_0, \delta, Q \setminus F \rangle$ is such that $L(\overline{A}) = \overline{L}$. This is because every accepting path in A is not accepting in $\overline{A}$, and vice versa.

Now that we know that the regular languages are closed under complementation, it is easy to show that they are closed under intersection: if L_1 and L_2 are regular languages, then $L_1 \cap L_2$ is also regular. This follows directly from fundamental theorems of set theory, since $L_1 \cap L_2$ can actually be written as $\overline{\overline{L_1} \cup \overline{L_2}}$, and we already know that the regular languages are closed under union and complementation. In fact, construction of an automaton for the intersection language is not very difficult, although it is less straight-forward than the previous examples.

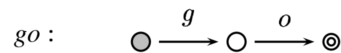
© 2001 by Shuly Wintner

2.6 Applications of finite-state automata in natural language processing

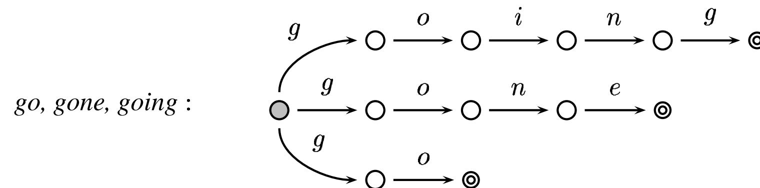
Finite-state automata are efficient computational devices for generating regular languages. An equivalent view would be to regard them as *recognizing* devices: given some automaton A and a word w, applying the automaton to the word yields an answer to the question: Is w a member of $L(A)$, the language accepted by the automaton? This reversed view of automata motivates their use for a simple yet necessary application of natural language processing: dictionary lookup.

Example 2.14 Dictionaries as finite-state automata

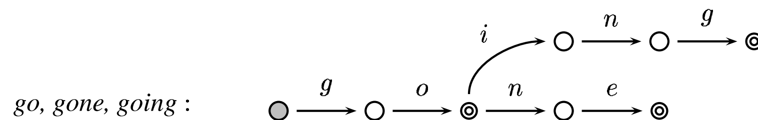
Many NLP applications require the use of lexicons or dictionaries, sometimes storing hundreds of thousands of entries. Finite-state automata provide an efficient means for storing dictionaries, accessing them and modifying their contents. To understand the basic organization of a dictionary as a finite-state machine, assume that an alphabet is fixed (we will use $\Sigma = \{a, b, \ldots, z\}$ in the following discussion) and consider how a single word, say *go*, can be represented. As we have seen above, a naïve representation would be to construct an automaton with a single path whose arcs are labeled by the letters of the word *go*:



To represent more than one word, we can simply add paths to our "lexicon", one path for each additional word. Thus, after adding the words *gone* and *going*, we might have:



This automaton can then be determinized and minimized:



With such a representation, a lexical lookup operation amounts to checking whether a word w is a member in the language generated by the automaton, which can be done by "walking" the automaton along the path indicated by w. This is an extremely efficient operation: it takes exactly one "step" for each letter of w. We say that the time required for this operation is *linear in the length of* w.

The organization of the lexicon as outlined above is extremely simplistic. The lexicon in this view is simply a list of words. For real application one is usually interested in associating certain information with every word in the lexicon. For simplicity, assume that we do not have to list a full dictionary entry with each word; rather, we only need to store some morpho-phonological information, such as the part-of-speech of the word, or its tense (in the case of verbs) or number (in the case of nouns). One way to achieve this goal is by extending the alphabet Σ: in addition to "ordinary" letters, Σ can include also special symbols, such as part-of-speech tags, morpho-phonological information, etc. An "analysis" of a (natural language) word w will in this case amount to recognition by the automaton of an extended word, w, followed by some special tags.

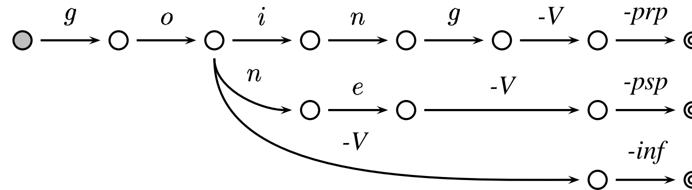
© 2001 by Shuly Wintner

Example 2.15 Adding morphological information to the lexicon

Suppose we want to add to the lexicon information about part-of-speech, and we use two tags: $-N$ for nouns, $-V$ for verbs. Additionally, we encode the number of nouns as *-sg* or *-pl*, and the tense of verbs as *-inf, -prp* or *-psp* (for infinitive, present participle and past participle, respectively). It is very important to note that the additional symbols are multi-character symbols: there is nothing in common to the alphabet symbol *-sg* and the sequence of two alphabet letters $\langle s,g \rangle$! In other words, the extended alphabet is:

$$\Sigma = \{a, b, c, \ldots, y, z, \text{-}N, \text{-}V, \text{-}sg, \text{-}pl, \text{-}inf, \text{-}prp, \text{-}psp\}$$

With the extended alphabet, we might construct the following automaton:



The language generated by the above automaton is no longer a set of words in English. Rather, it is a set of (simplisticly) "analyzed" strings, namely $\{$*go-V-inf, gone-V-psp, going-V-prp*$\}$.

Regular languages are particularly appealing for natural language processing for two main reasons. First, it turns out that most phonological and morphological processes can be straight-forwardly described using the operations that regular languages are closed under, and in particular concatenation. With very few exceptions (such as the interdigitation word-formation processes of Semitic languages or the duplication phenomena of some Asian languages), the morphology of most natural languages is limited to simple concatenation of affixes, with some morpho-phonological alternations, usually on a morpheme boundary. Such phenomena are easy to model with regular languages, and hence are easy to implement with finite-state automata. The other advantage of using regular languages is the inherent efficiency of processing with finite-state automata. Most of the algorithms one would want to apply to finite-state automata take time proportional to the length of the word being processed, independently of the size of the automaton. In computational terminology, this is called *linear time complexity*, and is as good as things can get.

In the next section we will see how two languages can be related such that one language contains only a set of natural language strings, while the other contains a set of analyzed strings, with the appropriate mapping from one language to the other.

2.7 Regular relations

We emphasized in the previous chapter a dual view of finite-state automata: as the mechanism is completely declarative, automata can be seen as either recognizing languages or generating them. In other words, depending on the application, one can use a finite-state automaton to generate the strings of its language or to determine whether a given string indeed belongs to this language. Finite-state automata are simply yet another way for defining (regular) languages, with the additional benefit that they are associated with efficient algorithms for manipulating such languages.

While this functionality is sufficient for some natural language applications, as was shown in section 2.6, sometimes it is useful to have a mechanism for *relating* two (formal) languages. For example, a part-of-speech tagger can be viewed as an application that relates a set of natural language strings (the *source* language) to a set of part-of-speech tags (the *target* language). A morphological analyzer can be viewed as a relation between natural language strings (the surface forms of words) and their internal structure (say, as sequences of morphemes). For dictionary lookup of some Semitic languages, one might want to relate

© 2001 by Shuly Wintner

surface forms with their root and pattern (as some dictionaries are indexed by root). In this section we discuss a computational device, very similar to finite-state automata, which defines a *relation* over two regular languages.

Example 2.16 Relations over languages

Consider a simple part-of-speech tagger: an application which associates with every word in some natural language a tag, drawn from a finite set of tags. In terms of formal languages, such an application implements a relation over two languages. For simplicity, assume that the natural language is defined over $\Sigma_1 = \{a, b, \ldots, z\}$ and that the set of tags is $\Sigma_2 = \{$*PRON, V, DET, ADJ, N, P*$\}$. Then the part-of-speech relation might contain the following pairs, depicted here vertically (that is, a string over Σ_1 is depicted over an element of Σ_2:

I	know	some	new	tricks	said	the	Cat	in	the	Hat
PRON	V	DET	ADJ	N	V	DET	N	P	DET	N

As another example, assume that Σ_1 is as above, and Σ_2 is a set of part-of-speech and morphological tags, including $\{$*-PRON, -V, -DET, -ADJ, -N, -P, -1, -2, -3, -sg, -pl, -pres, -past, -def, -indef*$\}$. A morphological analyzer is basically an application defining a relation between a language over Σ_1 and a language over Σ_2. Some of the pairs in such a relation are (vertically):

I	know	some	new	tricks	
I-PRON-1-sg	know-V-pres	some-DET-indef	new-ADJ	trick-N-pl	
said	the	Cat	in	the	Hat
say-V-past	the-DET-def	cat-N-sg	in-P	the-DET-def	hat-N-sg

Finally, consider the relation that maps every English noun in singular to its plural form. While the relation is highly regular (namely, adding "s" to the singular form), some nouns are irregular. Some instances of this relation are:

cat	hat	ox	child	mouse	sheep	goose
cats	hats	oxen	children	mice	sheep	geese

With the terminology we introduced so far we can define relations such as exemplified above in a simple way. A relation is defined over *two* alphabets, Σ_1 and Σ_2 (the second alphabet is sometimes denoted Δ). Of course, the two alphabets can be identical, but for many natural language applications they differ. A relation in $\Sigma^* \times \Sigma^*$ is regular if and only if its projections on both coordinates are regular languages. Informally, a regular relation is a set of pairs, each of which consists of one string over Σ_1 and one string over Σ_2, such that both the set of strings over Σ_1 and those over Σ_2 constitute regular languages.

2.8 Finite-state transducers

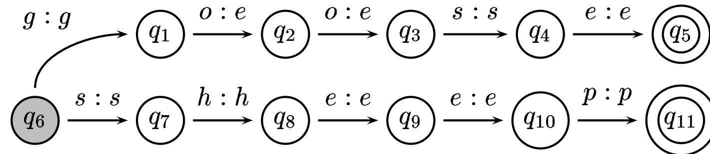
In the previous chapter we introduced finite-state automata as a computational device for defining regular languages. In a very similar way, *finite-state transducers* are a computational device for defining regular relations. Transducers are very similar to automata: the most significant difference is that the arcs are not labeled by single letters, but rather by pairs of symbols: one symbol from Σ_1 and one symbol from Σ_2. Formally, a finite-state transducer is a six-tuple $\langle Q, q_0, \Sigma_1, \Sigma_2, \delta, F\rangle$. Similarly to automata, Q is a finite set of states, q_0 is an element of Q, the initial state, F is a subset of Q, the final (or accepting) states, Σ_1 and Σ_2 are alphabets: finite sets of symbols, not necessarily disjoint (or different). Except for the addition of Σ_2, the only difference between automata and transducers is the transition relation δ: while for automata, δ relates

© 2001 by Shuly Wintner

states and alphabet symbols to states, in our case δ relates a state and *two* alphabet symbols, one from each alphabet, to a new state. Formally, δ is a relation over Q, Σ_1, Σ_2 and Q, or a subset of $Q \times \Sigma_1 \times \Sigma_2 \times Q$.

Example 2.17 Finite-state transducers

Following is a finite-state transducer relating the singular forms of two English words with their plural form. In this case, both alphabets are identical: $\Sigma_1 = \Sigma_2 = \{a, b, \ldots, z\}$. The set of nodes is $Q = \{q_1, q_2, \ldots, q_{11}\}$, the initial state is q_6 and the set of final states is $F = \{q_5, q_{11}\}$. The transitions from one state to another are depicted as labeled arcs; each arc bears two symbols, one from Σ_1 and one from Σ_2, separated by a colon (:). So, for example, $\langle q_1, o, e, q_2 \rangle$ is an element of δ.



Observe that each path in this device defines *two* strings: a concatenation of the left-hand side labels of the arcs, and a concatenation of the right-hand side labels. The upper path of the above transducer thus defines the pair *goose:geese*, whereas the lower path defines the pair *sheep:sheep*.

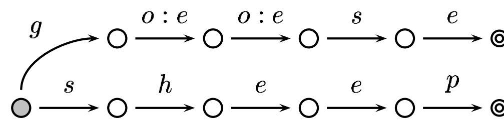
What constitutes a computation with a transducer? Similarly to the case of automata, a computation amounts to "walking" a path of the transducer, starting from the initial state and ending in some final state. Along the path, arcs bear bi-symbol labels: we can view the left-hand side symbol as an "input" symbol and the right-hand side symbol as an "output" symbol. Thus, each path of the transducer defines a pair of strings, an input string (over Σ_1) and an output string (over Σ_2). This pair of strings is a member of the relation defined by the transducer.

Formally, to define the relation induced by a transducer we must extend the relation δ from single transitions to paths, as we did for automata. The extended relation, $\hat{\delta}$, defined over Q, ${\Sigma_1}^*$, ${\Sigma_2}^*$ and Q, is defined as follows: for each $q \in Q$, $\hat{\delta}(q, \epsilon, \epsilon, q)$ holds, that is, each state is related to itself through an $\epsilon : \epsilon$ path; then, if $\hat{\delta}(q_1, w_1, w_2, q_2)$ and $\delta(q_2, a, b, q_3)$, then $\hat{\delta}(q_1, w_1 \cdot a, w_2 \cdot b, q_3)$ holds: if there is a path labeled $w_1 : w_2$ from q_1 to q_2 and an arc labeled $a : b$ from q_2 to q_3, then there is also a path labeled $w_1 \cdot a : w_2 \cdot b$ from q_1 to q_3. Finally, a pair $w_1 : w_2$ is *accepted* (or *generated*) by the transducer if and only if $\hat{\delta}(q_0, w_1, w_2, w_f)$ holds for some final state $q_f \in F$. The relation defined by the transducer is the set of all the pairs it accepts.

As a shorthand notation, when an arc is labeled by two identical symbols, we depict only one of them and omit the colon. As we did with automata, we usually suppress the identities of the nodes.

Example 2.18 Shorthand notation for transducers

With the above conventions, the transducer of example 2.17 is depicted as:



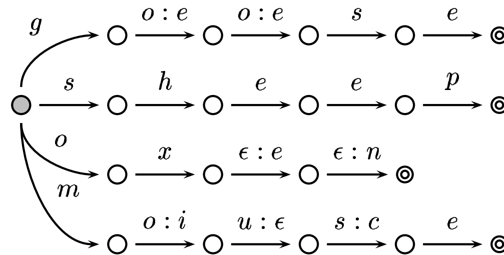
Of course, the above definition of finite-state transducers is not very useful: since each arc is labeled by exactly one symbol of Σ_1 and exactly one symbol of Σ_2, any relation that is implemented by such a transducer must relate only strings of exactly the same length. This should not be the case, and to overcome this limitation we extend the definition of δ to allow also ϵ-labels. In the extended definition, δ is a relation

© 2001 by Shuly Wintner

over Q, $\Sigma_1 \cup \{\epsilon\}$, $\Sigma_2 \cup \{\epsilon\}$ and Q. Thus a transition from one state to another can involve "reading" a symbol of Σ_1 without "writing" any symbol of Σ_2, or the other way round.

Example 2.19 Finite-state transducer with ϵ-labels

With the extended definition of transducers, we depict below an expanded transducer for singular–plural noun pairs in English.



Note that ϵ-labels can occur on the left or on the right of the ':' separator. The pairs accepted by this transducer are *goose:geese, sheep:sheep, ox:oxen* and *mouse:mice*.

It is worth noting that there exist variants of finite-state transducers in which the arcs are labeled not by single alphabet symbols but rather by strings thereof. For simplicity, we will only consider transducers as presented in this section.

2.9 Properties of regular relations

In the previous sections we discussed several properties of finite-state automata. The main observation was that finite-state automata generate exactly the set of regular languages; we also listed a few closure properties of such automata. The extension of automata to transducers carries with it some interesting results. First and foremost, finite-state transducers define exactly the set of regular relations. Many of the closure properties of automata are valid for transducers, but some are not. As these properties bear not only theoretical but also practical significance, we discuss them in more detail in this section.

Given some transducer T, consider what happens when the labels on the arcs of T are modified such that only the left-hand symbol remains. In other words, consider what is obtained when the transition relation δ is projected on three of its coordinates: Q, Σ_1 and Q only, ignoring the Σ_2 coordinate. It is easy to see that a finite-state automaton is obtained. We call this automaton the *projection* of T to Σ_1. In the same way, we can define the projection of T to Σ_2 by ignoring Σ_1 in the transition relation. Since both projections yield finite-state automata, they induce regular languages. Therefore the relation defined by T is a regular relation.

We can now consider certain operations on regular relations, inspired by similar operations on regular languages. For example, *union* is very easy to define. Recall that a regular relation is a subset of the Cartesian product of $\Sigma_1^* \times \Sigma_2^*$, that is, a set of pairs. If R_1 and R_2 are regular relations, then $R_1 \cup R_2$ is defined, and it is straight-forward to show that it is a regular relation. To define the union operation directly over transducers, we can simply extend the technique delineated in section 2.5 (see example 2.13), namely add a new initial state with two arcs labeled $\epsilon : \epsilon$ leading from it to the initial states of the given transducers.

In a similar way, the operation of *concatenation* can be extended to regular relations: if R_1 and R_2 are regular relations then $R_1 \cdot R_2 = \{\langle w_1 \cdot w_2, w_3 \cdot w_4\rangle \mid \langle w_1, w_3\rangle \in R_1 \textit{ and } \langle w_2, w_4\rangle \in R_2\}$. Again, the technique delineated in example 2.12 can be straight-forwardly extended to the case of transducers, and it is easy to show that $R_1 \cdot R_2$ is a regular relation.

© 2001 by Shuly Wintner

Example 2.20 Operations on finite-state transducers

Let R_1 be the following relation, mapping some English words to their German counterparts:

$R_1 = \{$*tomato:Tomate, cucumber:Gurke, grapefruit:Grapefruit, pineapple:Ananas, coconut:Koko*$\}$

Let R_2 be a similar relation: $R_2 = \{$*grapefruit:pampelmuse, coconut:Kokusnuß*$\}$. Then

$R_1 \cup R_2 = \{$*tomato:Tomate, cucumber:Gurke, grapefruit:Grapefruit, grapefruit:pampelmuse, pineapple:Ananas, coconut:Koko ,coconut:Kokusnuß*$\}$

A rather surprising fact is that regular relations are *not* closed under *intersection*. In other words, if R_1 and R_2 are two regular relations, then it very well might be the case that $R_1 \cap R_1$ is not a regular relation. It will take us beyond the scope of the material covered so far to explain this fact, but it is important to remember it when dealing with finite-state transducers. For this reason exactly it follows that the class of regular relations is not closed under *complementation*: since intersection can be expressed in terms of union and complementation, if regular relations were closed under complementation they would have been closed also under intersection, which we know is not the case.

A very useful operation that is defined for transducers is *composition*. Intuitively, a transducer relates one word ("input") with another ("output"). When we have more than one transducer, we can view the output of the first transducer as the input to the second. The composition of T_1 and T_2 relates the input language of T_1 with the output language of T_2, bypassing the intermediate level (which is the output of T_1 and the input of T_2). Formally, if R_1 is a relation from Σ_1^* to Σ_2^* and R_2 is a relation from Σ_2^* to Σ_3^* then the composition of R_1 and R_2, denoted $R_1 \circ R_2$, is a relation from Σ_1^* to Σ_3^* defined as $\{\langle w_1, w_3\rangle \mid$ *there exists a string* $w_2 \in \Sigma_2^*$ *such that* $w_1 R_1 w_2$ *and* $w_2 R_2 w_3\}$.

Example 2.21 Composition of finite-state transducers

Let R_1 be the following relation, mapping some English words to their German counterparts:

$R_1 = \{$*tomato:Tomate, cucumber:Gurke, grapefruit:Grapefruit, grapefruit:pampelmuse, pineapple:Ananas, coconut:Koko ,coconut:Kokusnuß*$\}$

Let R_2 be a similar relation, mapping French words to their English translations:

$R_2 = \{$*tomate:tomato, ananas:pineapple, pampelmousse:grapefruit, concombre:cucumber, cornichon:cucumber, noix-de-coco:coconut*$\}$

Then $R_2 \circ R_1$ is a relation mapping French words to their German translations (the English translations are used to compute the mapping, but are not part of the final relation):

$R_2 \circ R_1 = \{$*tomate:Tomate, ananas:Ananas, pampelmousse:Grapefruit, pampelmousse:Pampelmuse, concombre:Gurke, cornichon:Gurke, noix-de-coco:Koko, noix-de-coco:Kokusnuße*$\}$

Further reading

A very good formal exposition of regular languages and the computing machinery associated with them is given by Hopcroft and Ullman (1979, chapters 2–3). Another useful source is Partee, ter Meulen, and Wall (1990, chapter 17). For natural language applications of finite-state technology refer to Roche and Schabes (1997a), which is a collection of papers ranging from mathematical properties of finite-state machinery to

© 2001 by Shuly Wintner

linguistic modeling using them. The introduction (Roche and Schabes, 1997b) can be particularly useful, as will be Karttunen (1991). Koskenniemi (1983) is the classic presentation of *Two-Level Morphology*, and an exposition of the two-level rule formalism, which is demonstrated by an application of finite-state techniques to the morphology of Finnish. Kaplan and Kay (1994) is a classic work that sets the very basics of finite-state phonology, referring to automata, transducers and two-level rules. As an example of an extended regular expression language, with an abundance of applications to natural language processing, see Beesley and Karttunen (Forthcoming). Finally, Karttunen et al. (1996) is a fairly easy paper that relates regular expressions and relations to finite automata and transducers, and exemplifies their use in several language engineering applications.

© 2001 by Shuly Wintner

© 2001 by Shuly Wintner

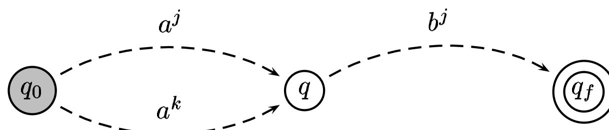
Chapter 3

Context-free grammars and languages

3.1 Where regular languages fail

Regular languages and relations were proven very useful for various applications of natural language processing, as the previous chapters demonstrate. Applications that require relatively "shallow" linguistic knowledge, especially around the finite domains of linguistics (e.g., morphology and phonology), such as dictionary lookup, morphological analysis and generation, part-of-speech tagging, etc., are easily modeled with finite-state techniques. However, there is a limit to what can be achieved with such means. The theoretical reason to this limit lies in the limited expressiveness of regular languages. We mentioned in passing that not *all* languages over Σ are regular, but we did not hint at what kind of languages lie beyond the regular ones. This is the topic of this section.

To exemplify a non-regular language, consider a simple language over the alphabet $\Sigma = \{a, b\}$ whose members are strings that consist of some number, n, of 'a's, followed by the same number of 'b's. Formally, this is the language $L = \{a^n \cdot b^n \mid n > 0\}$. Note that this is very different from the language $\{a^*b^*\}$: for a string to be a member of L, the number of a's must be identical to the number of b's. Now let us assume that this language is regular, and therefore that a deterministic finite-state automaton A exists whose language is L. Consider the language $L_i = \{a^i \mid i > 0\}$. Since every string in this language is a prefix of some string $(a^i \cdot b^i)$ of L, there must be a path in A starting from the initial state for every string in L_i. Of course, there is an infinite number of strings in L_i, but by its very nature, A has a finite number of states. Therefore there must be two different strings in L_i that lead the automaton to a single state. In other words, there exist two strings, a^j and a^k, such that $j \neq k$ but $\hat{\delta}(q_0, a^j) = \hat{\delta}(q_0, a^k)$. Let us call this state q. There must be a path labeled b^j leading from q to some final state q_f, since the string $a^j b^j$ is in L. This situation is schematically depicted below:



Therefore, there is also an accepting path $a^k b^j$ in A, and hence also $a^k b^j$ is in L, in contradiction to our assumption. Since our assumption leads to a contradiction, it must be false, and hence no deterministic finite-state automaton exists whose language is L.

We have seen one language, namely $L = \{a^n \cdot b^n \mid n > 0\}$, which cannot be defined by a finite-state automaton and therefore is not regular. In fact, there are several other such languages, and there is a well known technique, the so called *pumping lemma*, for proving that certain languages are not regular. If a language is not regular, then it cannot be denoted by a regular expression. We must look for alternative means of specification for non-regular languages.

3.2 Grammars

In order to specify a class of more complex languages, we introduce the notion of a *grammar*. Intuitively, a grammar is a set of rules which manipulate symbols. We distinguish between two kinds of symbols: *terminal* ones, which should be thought of as elements of the target language, and *non-terminal* ones, which are auxiliary symbols that facilitate the specification. It might be instructive to think of the non-terminal symbols as *syntactic categories*, such as *Sentence, Noun Phrase* or *Verb Phrase*. However, formally speaking, non-terminals are simply a set of symbols, and they have no "special", external interpretation where formal languages are concerned. Similarly, terminal symbols might correspond to letters of some natural language, or to words, or to nothing at all, when the grammar is of some non-natural language. Terminal symbols are simply elements of some finite set.

Rules can express the internal structure of "phrases", which should not necessarily be viewed as natural language phrases. Rather, they induce an internal structure on strings of the language, but this structure can be arbitrary, and does not have to be motivated by anything more than the convenience of expressing the required language. A rule is a non-empty sequence of symbols, a mixture of terminals and non-terminals, with the only requirement that the first element in the sequence be a non-terminal one. We write such rules with a special symbol, '$\rightarrow$', separating the distinguished leftmost non-terminal from the rest of the sequence. The leftmost non-terminal is sometimes referred to as the *head* of the rule, while the rest of the symbols are called the *body* of the rule.

Example 3.1 Rules

Assume that the set of terminals is $\{$*the, cat, in, hat*$\}$ and the set of non-terminals is $\{$*D, N, P, NP, PP*$\}$. Then possible rules over these two sets include:

$$\begin{array}{ll} D \rightarrow the & NP \rightarrow D\ N \\ N \rightarrow cat & PP \rightarrow P\ NP \\ N \rightarrow hat & NP \rightarrow NP\ PP \\ P \rightarrow in & \end{array}$$

Note that the terminal symbols correspond to words of English, and not to letters as was the case in the previous chapter.

Consider the rule $NP \rightarrow D\ N$. If we interpret *NP* as the syntactic category *noun phrase*, *D* as *determiner* and *N* as *noun*, then what the rule informally means is that one possible way to construct a noun phrase is by concatenating a determiner with a noun. More generally, a rule specifies one possible way to construct a "phrase" of the category indicated by its head: this way is by concatenating phrases of the categories indicated by the elements in the body of the rule. Of course, there might be more than one way to construct a phrase of some category. For example, there are two rules which define the structure of the category *NP* in example 3.1: either by concatenating a phrase of category *D* with one of category *N*, or by concatenating an *NP* with a *PP*.

In example 3.1, rules are of two kinds: the ones on the left have a single terminal symbol in their body, while the ones on the right have one or more non-terminal symbols, but no rule mixes both terminal and non-terminal symbols in its body. While this is a common practice when grammars for natural languages are concerned, nothing in the formalism requires such a format for rules. Indeed, rules can mix any combination of terminal and non-terminal symbols in their bodies. However, we will keep the convention of formatting rules in grammars for natural languages in this way.

A grammar is simply a finite set of rules. Since we must specify the alphabet and the set of non-terminals over which a particular grammar is defined, we say that, formally, a grammar is a four-tuple $G = \langle V, \Sigma, P, S \rangle$, where V is a finite set of non-terminal symbols, Σ is an alphabet of terminal symbols, P is a set of rules and S is the *start symbol*, a distinguished member of V. The rules (members of P) are

© 2001 by Shuly Wintner

sequences of terminals and non-terminals with a distinguished first element which is a non-terminal, that is, members of $V \times (V \cup \Sigma)^*$.

Note that this definition permits rules with empty bodies. Such rules, which consist of a left-hand side only, are called ϵ-rules, and are useful both for formal and for natural languages. Example 3.11 below will make use of an ϵ-rule.

Example 3.2 Grammar

The set of rules depicted in example 3.1 can constitute the basis for a grammar $G = \langle V, \Sigma, P, S \rangle$, where $V = \{$*D, N, P, NP, PP*$\}$, $\Sigma = \{$*the, cat, in, hat*$\}$, P is the set of rules and the start symbol S is *NP*.

In the sequel we will depict grammars by listing their rules only, as we did in example 3.1. We will keep a convention of using uppercase letters for the non-terminals and lowercase letters for the terminals, and we will assume that the set of terminals is the smallest that includes all the terminals mentioned in the rules, and the same for the non-terminals. Finally, we will assume that the start symbol is the head of the first rule, unless stated otherwise.

So far we talked about "rules" and "grammars" in general. In fact, formal language theory defines rules and grammars in a much broader way than that which was discussed above, and our definition is actually only a special case of rules and grammars. For various reasons that have to do with the format of the rules, this special case is known as *context-free* rules. This has nothing to do with the ability of grammars to refer to context; the term should not be taken mnemonically. However, although the term can be quite confusing, especially for natural language applications, it is well established in formal language theory and we will therefore use it here. In the next chapter we will present other rule systems and show that context-free rules and grammars indeed constitute a special case. In this chapter, however, we use the terms "rule" and "context-free rule" interchangeably, as we do for grammars, derivations etc.

3.3 Derivation

Formally speaking, each non-terminal symbol in a grammar denotes a language. For a simple rule such as *N* $\rightarrow$ *cat* we say that the language denoted by *N* includes the symbol *cat*. For a more complex rule such as *NP* $\rightarrow$ *D N*, the language denoted by *NP* contains the concatenation of the language denoted by *D* with the one denoted by *N*. Formally, for this rule we say that $L(NP) \supseteq L(D) \cdot L(N)$. Matters become more complicated when we consider a rule such as *NP* $\rightarrow$ *NP PP*. This is a *recursive* rule: a single non-terminal occurs both in the head and in the body of the rule, and hence we must define the language denoted by *NP* in terms of itself! While this might look strange, it is in fact well-defined and solutions for such recursive definitions are available.

In order to define the language denoted by a grammar symbol we need to define the concept of *derivation*. Derivation is a relation that holds between two *forms*, each a sequence of grammar symbols. Formally, given a grammar $G = \langle V, \Sigma, P, S \rangle$, we define the set of forms to be $(V \cup \Sigma)^*$: the set of all sequences of terminal and non-terminal symbols. We say that a form α *derives* a form β, denoted by $\alpha \Rightarrow \beta$, if and only if $\alpha = \gamma_l A \gamma_r$ and $\beta = \gamma_l \gamma_c \gamma_r$ and $A \rightarrow \gamma_c$ is a rule in P. A is called the *selected symbol*. Informally, this means that α derives β if a single non-terminal symbol, A, occurs in α, such that whatever is to its left in α, the (possibly empty) sequence of terminal and non-terminal symbols γ_l, occurs at the leftmost edge of β; and whatever is to the right of A in α, namely the (possibly empty) sequence of symbols γ_r, occurs at the rightmost edge of β; and the remainder of β, namely γ_c, constitutes the body of some grammar rule of which A is the head.

© 2001 by Shuly Wintner

Example 3.3 Derivation

Let G be the grammar of example 3.1, with *NP* the start symbol. The set of non-terminals of G is $V = \{D, N, P, NP, PP\}$ and the set of terminals is $\Sigma = \{the, cat, in, hat\}$. The set of forms therefore contains all the (infinitely many) sequences of elements from V and Σ, such as $\langle\rangle$, $\langle NP\rangle$, $\langle D\ cat\ P\ D\ hat\rangle$, $\langle D\ N\rangle$, $\langle the\ cat\ in\ the\ hat\rangle$, etc.

Let us start with a simple form, $\langle NP\rangle$. Observe that it can be written as $\gamma_l NP \gamma_r$, where both γ_l and γ_r are empty. Observe also that *NP* is the head of some grammar rule: the rule $NP \rightarrow D\ N$. Therefore, the form is a good candidate for derivation: if we replace the selected symbol *NP* with the body of the rule, while preserving its environment, we get $\gamma_l D\ N \gamma_r = D\ N$. Therefore, $\langle N\rangle \Rightarrow \langle D\ N\rangle$.

We now apply the same process to $\langle D\ N\rangle$. This time the selected symbol is *D* (we could have selected *N*, of course). The left context is again empty, while the right context is $\gamma_r = N$. As there exists a grammar rule whose head is *D*, namely $D \rightarrow the$, we can replace the rule's head by its body, preserving the context, and obtain the form $\langle the\ N\rangle$. Hence $\langle D\ N\rangle \Rightarrow \langle the\ N\rangle$.

Given the form $\langle the\ N\rangle$, there is exactly one non-terminal that we can select, namely *N*. However, there are two rules that are headed by *N*: $N \rightarrow cat$ and $N \rightarrow hat$. We can select either of these rules to show that both $\langle the\ N\rangle \Rightarrow \langle the\ cat\rangle$ and $\langle the\ N\rangle \Rightarrow \langle the\ hat\rangle$.

Since the form $\langle the\ cat\rangle$ consists of terminal symbols only, no non-terminal can be selected and hence it derives no form.

We now extend the derivation relation from a single step to an arbitrary number of steps. To formalize the concept of "any number of applications", one usually uses the *reflexive transitive closure* of a relation. In our case, the reflexive transitive closure of the derivation relation is a relation over forms, denoted '$\overset{*}{\Rightarrow}$', and defined recursively as follows: $\alpha \overset{*}{\Rightarrow} \beta$ if $\alpha \Rightarrow \beta$, or if $\alpha \Rightarrow \gamma$ and $\gamma \overset{*}{\Rightarrow} \beta$. Simply put, this definition means that the extended derivation relation '$\overset{*}{\Rightarrow}$' holds for two forms, α and β, either if α directly derives β, or if α directly derives some intermediate form, γ, which is known to (transitively) derive β.

Example 3.4 Extended derivation

In example 3.3 we showed that the following derivations hold:

$$\begin{array}{llll} (1) & \langle NP\rangle & \Rightarrow & \langle D\ N\rangle \\ (2) & \langle D\ N\rangle & \Rightarrow & \langle the\ N\rangle \\ (3) & \langle the\ N\rangle & \Rightarrow & \langle the\ cat\rangle \end{array}$$

Therefore, we trivially have:

$$\begin{array}{llll} (4) & \langle NP\rangle & \overset{*}{\Rightarrow} & \langle D\ N\rangle \\ (5) & \langle D\ N\rangle & \overset{*}{\Rightarrow} & \langle the\ N\rangle \\ (6) & \langle the\ N\rangle & \overset{*}{\Rightarrow} & \langle the\ cat\rangle \end{array}$$

From (2) and (6) we get

$$(7) \quad \langle D\ N\rangle \quad \overset{*}{\Rightarrow} \quad \langle the\ cat\rangle$$

and from (1) and (7) we get

$$(8) \quad \langle NP\rangle \quad \overset{*}{\Rightarrow} \quad \langle the\ cat\rangle$$

From here on we use the term "derivation" in reference to the extended, transitive relation; the original relation will be referred to as immediate derivation. The derivation relation is the basis for defining the language denoted by a grammar symbol. Consider the form obtained by taking a single grammar symbol, say $\langle A\rangle$; if this form derives a sequence of terminals, say $\langle a_1, \ldots a_n\rangle$, then we say that the terminal sequence (which is a string of alphabet symbols) is a member of the language denoted by A. Formally, the language of

© 2001 by Shuly Wintner

some non-terminal $A \in V$, denoted $L(A)$, is $\{a_1 \cdots a_n \mid a_i \in \Sigma \textit{ for } 1 \leq i \leq n \textit{ and } \langle A \rangle \overset{*}{\Rightarrow} \langle a_1, \ldots, a_n \rangle\}$. That is, the language denoted by A is the set of strings of terminal symbols that are derived by $\langle A \rangle$. The language of a grammar G, $L(G)$, is the language denoted by its start symbol, $L(S)$. When a string w is in $L(G)$ we say that w is *grammatical*.

Example 3.5 Language of a grammar

Consider again the grammar $G = \langle V, \Sigma, P, S \rangle$ where $V = \{D, N, P, NP, PP\}$, $\Sigma = \{the, cat, in, hat\}$, the start symbol S is *NP* and P is the following set of rules:

D → *the*	*NP* → *D N*
N → *cat*	*PP* → *P NP*
N → *hat*	*NP* → *NP PP*
P → *in*	

It is fairly easy to see that the language denoted by the non-terminal symbol *D*, $L(D)$, is the singleton set $\{the\}$. Similarly, $L(P)$ is $\{in\}$ and $L(N) = \{cat, hat\}$. It is more difficult to define the languages denoted by the non-terminals *NP* and *PP*, although is should be straight-forward that the latter is obtained by concatenating $\{in\}$ with the former. We claim, without providing a proof, that $L(NP)$ is the denotation of the regular expression $(the \cdot (cat + hat) \cdot (in \cdot the \cdot (cat + hat))^*)$.

3.4 Derivation trees

The derivation relation is the basis for the definition of languages, but sometimes derivations provide more information than is actually needed. In particular, sometimes two derivations of the same string differ not in the rules that were applied but only in the order in which they were applied. Consider again the grammar of example 3.5. Starting with the form $\langle NP \rangle$ it is possible to derive the string *the cat* in two ways:

$$(1) \quad \langle NP \rangle \Rightarrow \langle D\ N \rangle \Rightarrow \langle D\ cat \rangle \Rightarrow \langle the\ cat \rangle$$
$$(2) \quad \langle NP \rangle \Rightarrow \langle D\ N \rangle \Rightarrow \langle the\ N \rangle \Rightarrow \langle the\ cat \rangle$$

Derivation (1) applies first the rule *N* → *cat* and then the rule *D* → *the* whereas derivation (2) applies the same rules in the reverse order. But since both use the same rules to derive the same string, it is sometimes useful to collapse such "equivalent" derivations into one. To this end the notion of *derivation trees* is introduced.

A derivation tree (sometimes called *parse* tree, or simply tree) is a visual aid in depicting derivations, and a means for imposing structure on a grammatical string. Trees consist of vertices and branches; a designated vertex, the *root* of the tree, is depicted on the top. Then, branches are simply connections between pairs of vertices. Intuitively, trees are depicted "upside down", since their root is at the top and their leaves are at the bottom. An example for a derivation tree for the string *the cat in the hat* with the grammar of example 3.5 is given in example 3.6 (page 32).

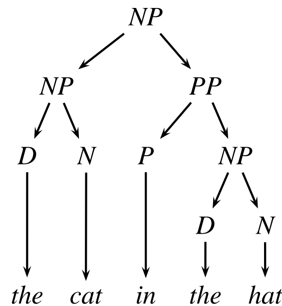
Formally, a tree consists of a finite set of vertices and a finite set of branches (or arcs), each of which is an ordered pair of vertices. In addition, a tree has a designated vertex, the *root*, which has two properties: it is not the target of any arc, and every other vertex is accessible from it (by following one or more branches). When talking about trees we sometimes use family notation: if a vertex v has a branch leaving it which leads to some vertex u, then we say that v is the *mother* of u and u is the *daughter*, or *child*, of v. If u has two daughters, we refer to them as *sisters*. Derivation trees are defined with respect to some grammar G, and must obey the following conditions:

1. every vertex has a *label*, which is either a terminal symbol, a non-terminal symbol or ϵ;

2. the label of the root is the start symbol;

© 2001 by Shuly Wintner

Example 3.6 Derivation tree



3. if a vertex v has an outgoing branch, its label must be a non-terminal symbol; furthermore, this symbol must be the head of some grammar rule; and the elements in the body of the same rule must be the labels of the children of v, in the same order;

4. if a vertex is labeled ϵ, it is the only child of its mother.

A *leaf* is a vertex with no outgoing branches. A tree induces a natural "left-to-right" order on its leaves; when read from left to right, the sequence of leaves is called the *frontier*, or *yield* of the tree.

By the way we defined them, derivation trees correspond very closely to derivations. In fact, it is easy to show that for a form α, a non-terminal symbol A derives α if and only if α is the yield of some parse tree whose root is A. In other words, whenever some string can be derived from a non-terminal, there exists a derivation tree for that string, with the same non-terminal as its root. However, sometimes there exist different derivations of the same string that correspond to a single tree. In fact, the tree representation collapses exactly those derivations that differ from each other only in the order in which rules are applied.

Example 3.7 Correspondence between trees and derivations

Consider the tree of example 3.6. Each non-leaf vertex in the tree corresponds to some grammar rule (since it must be labeled by the head of some rule, and its children must be labeled by the body of the same rule). For example, the root of the tree, which is labeled *NP*, corresponds to the rule *NP* $\rightarrow$ *NP PP*; the leftmost *D*-labeled vertex corresponds to the rule *D* $\rightarrow$ *the*. This tree represents the following derivations (among others):

(1) *NP* $\Rightarrow$ *NP PP* $\Rightarrow$ *D N PP* $\Rightarrow$ *D N P NP* $\Rightarrow$ *D N P D N* $\Rightarrow$ *the N P D N* $\Rightarrow$ *the cat P D N* $\Rightarrow$ *the cat in D N* $\Rightarrow$ *the cat in the N* $\Rightarrow$ *the cat in the hat*

(2) *NP* $\Rightarrow$ *NP PP* $\Rightarrow$ *D N PP* $\Rightarrow$ *the N PP* $\Rightarrow$ *the cat PP* $\Rightarrow$ *the cat P NP* $\Rightarrow$ *the cat in NP* $\Rightarrow$ *the cat in D N* $\Rightarrow$ *the cat in the N* $\Rightarrow$ *the cat in the hat*

(3) *NP* $\Rightarrow$ *NP PP* $\Rightarrow$ *NP P NP* $\Rightarrow$ *NP P D N* $\Rightarrow$ *NP P D hat* $\Rightarrow$ *NP P the hat* $\Rightarrow$ *NP in the hat* $\Rightarrow$ *D N in the hat* $\Rightarrow$ *D cat in the hat* $\Rightarrow$ *the cat in the hat*

While exactly the same rules are applied in each derivation (the rules are uniquely determined by the tree), they are applied in different orders. In particular, derivation (2) is a *leftmost* derivation: in every step the leftmost non-terminal symbol of a derivation is expanded. Similarly, derivation (3) is *rightmost*.

Sometimes, however, different derivations (of the same string!) correspond to different trees. This can happen only when the derivations differ in the rules which they apply. When more than one tree exists for some string, we say that the string is *ambiguous*. Ambiguity is a major problem when grammars are

© 2001 by Shuly Wintner

used for certain formal languages, in particular for programming languages. But for natural languages, ambiguity is unavoidable as it corresponds to properties of the natural language itself.

Example 3.8 Ambiguity

Consider again the grammar of example 3.5, and the following string:

the cat in the hat in the hat

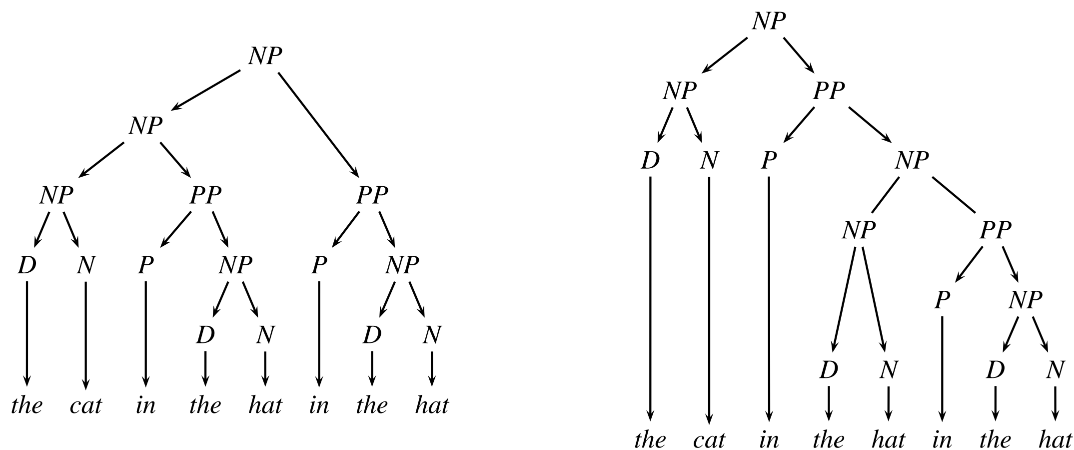
What does this sequence of words mean? Intuitively, there can be (at least) two readings for this string: one in which a certain cat wears a hat-in-a-hat, and one in which a certain cat-in-a-hat is inside a hat. If we wanted to indicate the two readings with parentheses, we would distinguish between

((the cat in the hat) in the hat)

and

(the cat in (the hat in the hat))

This distinction in intuitive meaning is reflected in the grammar, and hence two different derivation trees, corresponding to the two readings, are available for this string:



Using linguistic terminology, in the left tree the second occurrence of the prepositional phrase *in the hat* modifies the noun phrase *the cat in the hat*, whereas in the right tree it only modifies the (first occurrence of) the noun phrase *the hat*. This situation is known as *syntactic* or *structural* ambiguity.

3.5 Expressiveness

In example 3.5 above we presented a context-free grammar whose language, we claimed, is regular. However, recall from section 3.1 that the motivation behind the introduction of context-free grammars was the inability of regular expressions to denote certain languages (or, in other words, the limitations of the class of regular languages). We must therefore show that context-free grammars do provide the means for describing some non-regular languages.

In this section we will concentrate on a formal (that is, not natural) example, for the sake of simplicity. The language that was shown to be non-regular in section 3.1 is $L = \{a^n b^n \mid n > 0\}$, and this is the language for which we provide a context-free grammar here. The grammar, $G = \langle V, \Sigma, P, S \rangle$, has two

© 2001 by Shuly Wintner

terminal symbols: $\Sigma = \{a, b\}$, and three non-terminal symbols: $V = \{S, A, B\}$, with S the start symbol. The idea is to have S derive strings of one or more 'a's, and B – strings of 'b's. But of course, the grammar must somehow guarantee that the number of the 'a's matches the number of the 'b's. This is achieved by guaranteeing that whenever an A is added to some form, a B is also added.

Example 3.9 A context-free grammar for $L = \{a^n b^n \mid n > 0\}$

(1)	$S \rightarrow A\ S\ B$
(2)	$S \rightarrow a\ b$
(3)	$A \rightarrow a$
(4)	$B \rightarrow b$

The idea behind the grammar of example 3.9 is simple: the language of the non-terminal A is simply $\{a\}$, and similarly $L(B) = \{b\}$. The language of S is more complicated. With the second rule, $S \rightarrow a\ b$, S can derive the string $a\ b$. The first rule is recursive: it expands an occurrence of S in a form to $A\ S\ B$. Since the A will derive a and the B will derive b, this amounts to replacing an occurrence of S in a form with $a\ S\ b$. Note that whenever an a is added to the left of the S, a b is added on its right. This ensures that the number of the as matches the number of the bs. Eventually, the S in a form can be expanded into $a\ b$, to complete the derivation.

Example 3.10 Derivation with the grammar G

An example derivation with the grammar G of example 3.9 is given below. The derived string is $a\ a\ b\ b$, which is indeed in the language of G. The number above the derivation arrow corresponds to the number of the rule of G that was used for the derivation.

$$\begin{array}{lll} \langle S \rangle & \stackrel{1}{\Rightarrow} & \langle A\ S\ B \rangle \\ & \stackrel{2}{\Rightarrow} & \langle A\ a\ b\ B \rangle \\ & \stackrel{4}{\Rightarrow} & \langle A\ a\ b\ b \rangle \\ & \stackrel{3}{\Rightarrow} & \langle a\ a\ b\ b \rangle \end{array}$$

What if we were interested in a slightly different language, namely $L' = \{a^n b^n \mid n \geq 0\}$? Notice that the only difference between the two languages lies in the empty string, ϵ, which is a member of the new language. A grammar for L' is given in example 3.11 to demonstrate the use of what is know as ϵ-rules, that is, rules with empty bodies. There is nothing in the definition of a rule which requires that its body be non-empty. When an ϵ-rule participates in a derivation, whenever its head is the selected element in some form, it is replaced by the empty sequence, that is, the length of the form decreases.

Example 3.11 A context-free grammar for $L' = \{a^n b^n \mid n \geq 0\}$

(1)	$S \rightarrow A\ S\ B$
(2)	$S \rightarrow \epsilon$
(3)	$A \rightarrow a$
(4)	$B \rightarrow b$

This grammar is very similar to the grammar for L of example 3.9; the only difference is that the halting condition for the recursion, namely rule (2), now expands S to ϵ rather to $a\ b$. This allows for the empty string to be a member of the language: a derivation for ϵ simply starts with $\langle S \rangle$ and applies the second rule, yielding $\langle \rangle$.

© 2001 by Shuly Wintner

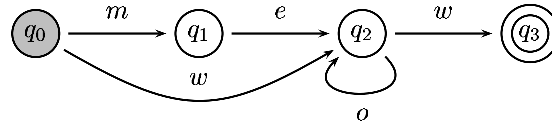
Since the language $L = \{a^n b^n \mid n > 0\}$ is not a regular language, and since we have shown above a context-free grammar for generating it, we have proven that some context-free languages are not regular languages. This brings up an interesting question: what is the relationship between these two classes of languages? There are only two possibilities: either the context-free languages properly contain the regular languages, or they simply overlap: they have a non-empty intersection, but each class contains members that are not members of the other class. It turns out that it is relatively simple to show that the former is the case: every regular language is also context-free. In other words, given some finite-state automaton which generates some language L, it is always possible to construct a context-free grammar whose language is L. The regular languages are properly contained in the context-free languages.

To validate the above claim, we conclude this section with a discussion of converting automata to context-free grammars. The rationalization is that if we can show, for any finite-state automaton A, a grammar G such that $L(A) = L(G)$, then we have proven that any regular language is context-free. In other words, in what follows we "simulate" a computation of a finite-state automaton by a derivation of a context-free grammar.

Let $A = \langle Q, q_0, \delta, F\rangle$ be a deterministic finite-state automaton with no ϵ-moves over the alphabet Σ. The grammar we define to simulate A is $G = \langle V, \Sigma, P, S\rangle$, where the alphabet Σ is that of the automaton, and where the set of non-terminals, V, is the set Q of the automaton states. In other words, each state of the automaton is translated to a non-terminal of the grammar. The idea is that a single (immediate) derivation step with the grammar will simulate a single arc traversal with the automaton. Since automata states are simulated by grammar non-terminals, it is reasonable to simulate the initial state by the start symbol, and hence the start symbol S is q_0. What is left, of course, are the grammar rules. These come in two varieties: first, for every automaton arc $\delta(q, a) = q'$ we stipulate a rule $q \rightarrow a\, q'$. Then, for every final state $q_f \in F$, we add the rule $q_f \rightarrow \epsilon$. This process is demonstrated for a small grammar in example 3.12.

Example 3.12 Simulating a finite-state automaton by a grammar

Consider the simple automaton $\langle Q, q_0, \delta, F\rangle$ depicted below, where $Q = \{q_0, q_1, q_2, q_3\}$, $F = \{q_3\}$ and δ is $\{\langle q_0, m, q_1\rangle, \langle q_1, e, q_2\rangle, \langle q_2, o, q_2\rangle, \langle q_2, w, q_3\rangle, \langle q_0, w, q_2\rangle\}$:



The grammar $G = \langle V, \Sigma, P, S\rangle$ which simulates this automaton has $V = \{q_0, q_1, q_2, q_3\}$, $S = q_0$ and the set of rules:

(1) $q_0 \rightarrow m\, q_1$
(2) $q_1 \rightarrow e\, q_2$
(3) $q_2 \rightarrow o\, q_2$
(4) $q_2 \rightarrow w\, q_3$
(5) $q_0 \rightarrow w\, q_2$
(6) $q_3 \rightarrow \epsilon$

The string *meoow*, for example, is generated by the automaton by walking along the path $q_0 - q_1 - q_2 - q_2 - q_2 - q_3$. The same string is generated by the grammar with the derivation

$$\langle q_0\rangle \stackrel{1}{\Rightarrow} \langle mq_1\rangle \stackrel{2}{\Rightarrow} \langle meq_2\rangle \stackrel{3}{\Rightarrow} \langle meoq_2\rangle \stackrel{3}{\Rightarrow} \langle meooq_2\rangle \stackrel{4}{\Rightarrow} \langle meoowq_3\rangle \stackrel{6}{\Rightarrow} \langle meoow\rangle$$

We will not provide a proof that the language of the grammar indeed coincides with the language of the automaton, but the validity of this proposition should be easy to see.

It is extremely important to realize that while for every finite-state automaton A a grammar G exists

© 2001 by Shuly Wintner

such that $L(A) = L(G)$, the reverse does *not* hold. There exist grammars whose language cannot be generated by any finite-state automaton. Our conclusion, then, is that the class of regular languages is properly contained within the class of context-free languages.

Observing the grammar of example 3.12, a certain property of the rules stands out: the body of each of the rules either consists of a terminal followed by a non-terminal or is empty. This is a special case of what are known as *right-linear* grammars. In a right-linear grammar, the body of each rule consists of a (possibly empty) sequence of terminal symbols, optionally followed by a single non-terminal symbol. Most importantly, no rule exists whose body contains more than one non-terminal; and if a non-terminal occurs in the body, it is in the final position. Right-linear grammars are a restricted variant of context-free grammars, and it can be shown that they generate all and only the regular languages.

3.6 Linguistic examples

Context-free grammars are a powerful device, both for the specification of formal (e.g., programming) languages and for describing the syntax (and, sometimes, also other aspects) of natural languages. One of the reasons for their appeal is their efficiency: while not as efficient to implement as finite-state automata, there exist algorithms that determine whether a given string is in the language of some grammar in time that is proportional to the cube of the length of the string. Compared to finite-state automata, where the time for such a task is proportional to the length of the string (that is, linear rather than cubic), this might seem a major difference; but given the extended expressiveness of context-free languages over regular languages, this is sometimes worth the price. In this section we exemplify some characteristic linguistic issues that can be handled elegantly with context-free grammars.

Let us start with the basic structure of sentences in English. The grammar of example 3.5 can be easily extended to cover verb phrases, in addition to noun phrases: all that is required is a set of rules that derive verbs, say from a non-terminal called *V*, and a few rules that define the structure of verb phrases:

V → *sleeps*	*VP* → *V*
V → *smile*	*VP* → *VP NP*
V → *loves*	*VP* → *VP PP*
V → *saw*	

Now all that is needed is a rule for sentence formation: *S* → *NP VP*, and the augmented grammar can derive strings such as *the cat sleeps* or *the cat in the hat saw the hat*. The grammar, along with an example derivation tree, are depicted in example 3.13 (page 37).

There are two major problems with the grammar of example 3.13. Firstly, it ignores the valence of verbs: there is no distinction among subcategories of verbs, and an intransitive verb such as *sleeps* might occur with a noun phrase complement, while a transitive verb such as *loves* might occur without one. In such a case we say that the grammar *overgenerates*: it generates strings that are not in the intended language. Secondly, there is no treatment of subject–verb agreement, so that a singular subject such as *the cat* might be followed by a plural form of verb such as *smile*. This is another case of overgeneration. Both problems are easy to solve.

To account for valence, we simply replace the non-terminal symbol *V* by a set of symbols: *Vtrans, Vintrans, Vditrans* etc. We also change the grammar rules accordingly:

VP → *Vintrans*	*Vintrnas* → *sleeps*
VP → *Vtrans NP*	*Vintrans* → *smile*
VP → *Vditrans NP PP*	*Vtrans* → *loves*
	Vditrans → *give*

To account for agreement, we can again extend the set of non-terminal symbols such that categories that must agree reflect in the non-terminal that is assigned for them the features on which they agree. In the very

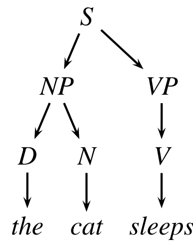
© 2001 by Shuly Wintner

Example 3.13 A context-free grammar for English sentences

The extended grammar is $G = \langle V, \Sigma, P, S\rangle$ where $V = \{$*D, N, P, NP, PP, V, VP, S*$\}$, $\Sigma = \{$*the, cat, in, hat, sleeps, smile, loves, saw*$\}$, the start symbol is S and P is the following set of rules:

$S \rightarrow NP\ VP$	$D \rightarrow the$
$NP \rightarrow D\ N$	$N \rightarrow cat$
$NP \rightarrow NP\ PP$	$N \rightarrow hat$
$PP \rightarrow P\ NP$	$P \rightarrow in$
$VP \rightarrow V$	$V \rightarrow sleeps$
$VP \rightarrow VP\ NP$	$V \rightarrow smile$
$VP \rightarrow VP\ PP$	$V \rightarrow loves$
	$V \rightarrow saw$

A derivation tree for *the cat sleeps* is:



simple case of English, it is sufficient to multiply the set of "nominal" and "verbal" categories, so that we get *Dsg, Dpl, Nsg, Npl, NPsg, NPpl, Vsg, Vlp, VPsg, VPpl* etc. We also change the set of rules accordingly:

$S \rightarrow NPsg\ VPsg$	$S \rightarrow NPpl\ VPpl$
$NPsg \rightarrow Dsg\ Nsg$	$NPpl \rightarrow Dpl\ Npl$
$NPsg \rightarrow NPsg\ PP$	$NPpl \rightarrow NPpl\ PP$
$PP \rightarrow P\ NP$	
$VPsg \rightarrow Vsg$	$VPpl \rightarrow Vpl$
$VPsg \rightarrow VPsg\ NP$	$VPpl \rightarrow VPpl\ NP$
$VPsg \rightarrow VPsg\ PP$	$VPpl \rightarrow VPpl\ PP$
$Nsg \rightarrow cat$	$Npl \rightarrow cats$
$Nsg \rightarrow hat$	$Npl \rightarrow hats$
$P \rightarrow in$	
$Vsg \rightarrow sleeps$	$Vpl \rightarrow sleep$
$Vsg \rightarrow smiles$	$Vpl \rightarrow smile$
$Vsg \rightarrow loves$	$Vpl \rightarrow love$
$Vsg \rightarrow saw$	$Vpl \rightarrow saw$
$Dsg \rightarrow a$	$Dpl \rightarrow many$

Context-free grammars can be used for a variety of syntactic constructions, including some non-trivial phenomena such as unbounded dependencies, extraction, extraposition etc. However, some (formal) languages are not context-free, and therefore there are certain sets of strings that cannot be generated by context-free grammars. The interesting question, of course, involves natural languages: are there natural languages that are not context-free? Are context-free grammars sufficient for generating every natural language? This is the topic we discuss in the next chapter.

© 2001 by Shuly Wintner

3.7 Formal properties of context-free languages

In section 2.9 we discussed some closure properties of regular languages and came to the conclusion that that class was closed under union, intersection, complementation, concatenation, Kleene-closure etc. These are useful properties, both for theoretical reasoning with regular languages and for practical applications. Unfortunately, context-free languages are no so "well-behaved". We discuss some of their closure properties in this section.

It should be fairly easy to see that context-free languages are closed under union. Given two context-free grammars $G_1 = \langle V_1, \Sigma_1, P_1, S_1 \rangle$ and $G_2 = \langle V_2, \Sigma_2, P_2, S_2 \rangle$, a grammar $G = \langle V, \Sigma, P, S \rangle$ whose language is $L(G_1) \cup L(G_2)$ can be constructed as follows: the alphabet Σ is the union of Σ_1 and Σ_2, the non-terminal set V is a union of V_1 and V_2, plus a new symbol S, which is the start symbol of G. Then, the rules of G are just the union of the rules of G_1 and G_2, with two additional rules: $S \rightarrow S_1$ and $S \rightarrow S_2$, where S_1 and S_2 are the start symbols of G_1 and G_2, respectively. Clearly, every derivation in G_1 can be simulated by a derivation in G using the same rules exactly, starting with the rule $S \rightarrow S_1$, and the same is true for derivations in G_2. Also, since S is a new symbol, no other derivations in G are possible. Therefore $L(G) = L(G_1) \cup L(G_2)$.

A similar idea can be used to show that the context-free languages are closed under concatenation: here we only need one additional rule, namely $S \rightarrow S_1\ S_2$, and the rest of the construction is identical. Any derivation in G will "first" derive a string of G_1 (through S_1) and then a string of G_2 (through S_2). To show closure under the Kleene-closure operation, use a similar construction with the added rules $S \rightarrow \epsilon$ and $S \rightarrow S\ S_1$.

However, it is possible to show that the class of context-free languages is not closed under intersection. That is, if L_1 and L_2 are context-free languages, then it is not guaranteed that $L_1 \cap L_2$ is context-free as well. From this fact it follows that context-free languages are not closed under complementation, either. While context-free languages are not closed under intersection, they *are* closed under intersection with regular languages: if L is a context-free language and R is a regular language, then it is guaranteed that $L \cap R$ is context-free.

Further reading

Context-free grammars and languages are discussed by Hopcroft and Ullman (1979, chapters 4, 6) and Partee, ter Meulen, and Wall (1990, chapter 18). A linguistic formalism that is based on the ability of context-free grammars to provide adequate analyses for natural languages is Generalized Phrase Structure Grammars, or GPSG (Gazdar et al., 1985).

© 2001 by Shuly Wintner

Chapter 4

The Chomsky hierarchy

4.1 A hierarchy of language classes

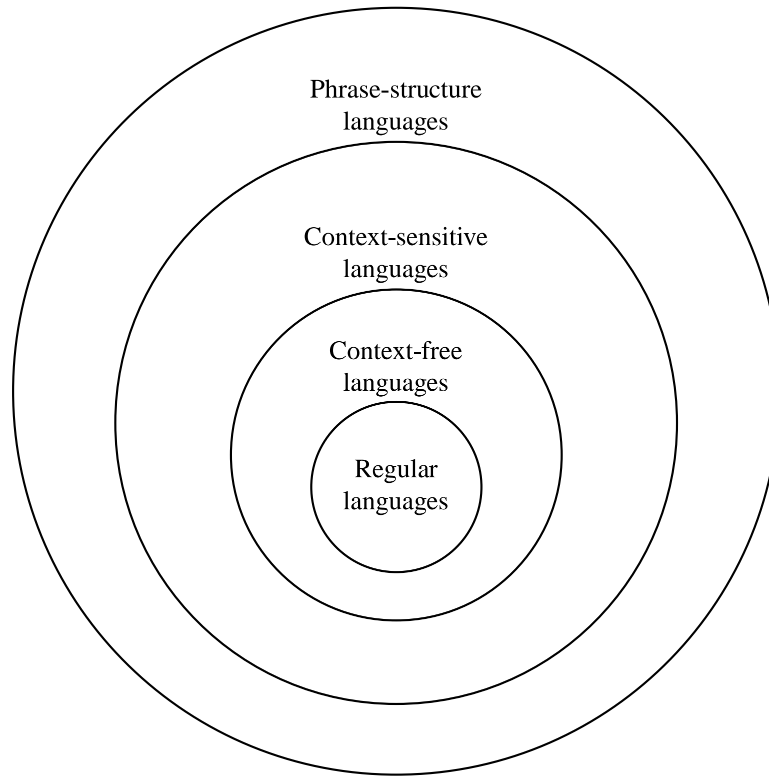
We have encountered so far three types of mechanisms for defining languages: expressions, automata and grammars. Let us focus on grammars in this chapter. We have seen two types of grammars: context-free grammars, which generate the set of all context-free languages; and right-linear grammars, which generate the set of all regular languages. Right-linear grammars are a special case of context-free grammars, where additional constraints are imposed on the form of the rules (in this case, at most one non-terminal symbol can be present in the body of a rule, and it must be the final element). The pattern that we emphasize here is that by placing certain constraints on the form of the rules, we can constrain the expressive power of the formalism. What would happen if we used this rationalization in the reverse direction, that is, allowed more freedom in the form of the rules?

One way to extend the expressiveness of grammars is to allow more than a single non-terminal symbol in the *head* of the rules. With context-free grammars, a rule can be applied (during a derivation) when its head is an element in a form. We can extend this formalism by referring, in the application of rules, to a certain context. While a context-free rule can specify that a certain non terminal, say A, expands to a certain sequence of terminals and non-terminals, say α, a rule in the extended formalism will specify that such an expansion is allowed only if the context of A in the form, that is, A's neighbors to the right and left, are as specified in the rule. Due to this reference to context, the formalism we describe is known as *context-sensitive* grammars. A rule in a context-sensitive grammar has the form $\alpha_1 \; A \; \alpha_2 \rightarrow \alpha_1 \beta \alpha_2$, where α_1, α_2 and β are all (possibly empty) sequences of terminal and non-terminal symbols. The other components of context-sensitive grammars are as in the case of context-free grammars. For example, a context-sensitive rule might expand a verb phrase to a ditransitive verb such as *love* only if it is preceded by a nominative noun phrase and followed by an accusative one: *NPnom VP NPacc* $\rightarrow$ *NPnom love NPacc*. This format of rules is mostly known from phonology, where phonological processes are usually described as "some phoneme A is realized as β when it is preceded by α_1 and followed by α_2".

As usual, the class of languages that can be generated by context-sensitive grammars is called the context-sensitive languages. Considering that every context-free grammar is a special case of context-sensitive grammars (with an empty context), it should be clear that every context-free language is also context-sensitive, or in other words, that the context-free languages are contained in the set of the context-sensitive ones. As it turns out, this containment is proper, that is, there are context-sensitive languages that are not context-free.

What we are finding is a hierarchy of classes of languages: the regular languages are properly contained in the context-free languages, which are properly contained in the context-sensitive languages. These, in turn, are known to be properly contained in the set of languages generated by the so-called *unrestricted* or

general phrase structure grammars. This hierarchy of language classes was first defined by Noam Chomsky, and is therefore called the *Chomsky Hierarchy of Languages*. It is schematically depicted below:



4.2 The location of natural languages in the hierarchy

The Chomsky Hierarchy of Languages reflects a certain order of complexity: in some sense, the lower the language class is in the hierarchy, the simpler are its possible constructions. This is a very informal notion, and indeed there is nothing sacred about this particular hierarchy: one could conceive of criteria other than the form of the rules for defining language classes, and get at a different hierarchy altogether. However, given the significance of the Chomsky hierarchy in formal language theory and computer science in general, it is interesting to investigate the location of formal languages with respect to this hierarchy. In other words, where does the class of natural languages fall? Are there natural languages that are trans-regular (cannot be described by a regular expression)? Are there any that are trans-context-free (cannot be described by a context-free grammar)? Beyond the expressive power of even context-sensitive grammars?

In his 1957 *Syntactic Structures* (Chomsky, 1957), Noam Chomsky presents a theorem that says "English is not a regular language"; as for context-free languages, Chomsky says "I do not know whether or not English is itself literally outside the range of such analyses". Incredibly, it was for many years well accepted that natural languages stood beyond the expressive power of context-free grammars, with not a single well established proof. To illustrate this situation, here are a few quotes from introductory texts in linguistics.

In one of the most popular syntax textbooks, *An introduction to the principles of transformational syntax* (Akmajian and Heny, 1976, pp. 84-86), in a chapter titled "The need for transformations", the authors present a small fragment of English that deals with auxiliary constructions. They present a small CFG for the fragment, show that it undergenerates, then amend it and show that the amended grammar overgenerates.

© 2001 by Shuly Wintner

Finally, they modify the grammar again, and show that it generates the required language, but does not "express linguistically significant generalizations about the language". They conclude:

> Since there seem to be no way of using such PS rules to represent an obviously significant generalization about one language, namely, English, we can be sure that phrase structure grammars cannot possibly represent ALL the significant aspects of language structure. We must introduce a new kind of rule that will permit us to do so.

The logic that Akmajian and Heny are applying is the following: try to list as many grammars as possible for a given language; if you can't find one that is context-free, the language is trans-context-free. But a natural language, and indeed any language, will have an infinite number of grammars. It is logically impossible to search through all of them for one that is context-free. Hence this technique of demonstration by enumeration of candidate grammars, in order to show that they are inappropriate, is out of the question.

In a similar way, Peter Culicover in "Syntax" (Culicover, 1976) presents a sequence of infinitely many sentences with an increasing degree of embedding, and a series of CF rules that manage to capture increasingly more complex constructions. He concludes (p. 28):

> In general, for any phrase structure grammar containing a finite number of rules like (2.5), (2.52) and (2.54) it will always be possible to construct a sentence that the grammar will not generate. In fact, because of recursion there will always be an infinite number of such sentences. Hence, the phrase structure analysis will not be sufficient to generate English.

Later on (p. 50) he explores a phenomenon of affix hoping in auxiliary constructions in English, and concludes that "this makes it impossible to capture the observed generalizations in purely phrase structure terms". Again, no proof is given.

What is the source for this confusion? Perhaps the reason lies in the notion of "context-freeness"; one might be misled to believe that context-free grammars cannot capture phenomena that involve context. This misconception runs so deep, that you find incredible statements such as in *Transformational grammar* (Grinder and Elgin, 1973). The authors present (p. 56) what must be the simplest context-free grammar for a small subset of English, and demonstrate that while it correctly generates *the girl saw the boy*, it also wrongly generates *the girl kiss the boy*. They claim that "this well-known syntactic phenomenon demonstrates clearly the inadequacy of ... context-free phrase-structure grammars". More precisely, they state (p. 58) that:

> The defining characteristic of a context-free rule is that the symbol to be rewritten is to be rewritten without reference to the context in which it occurs... Thus, by definition, one cannot write a context-free rule that will expand the symbol V into *kiss* in the context of being immediately preceded by the sequence *the girls* and that will expand the symbol V into *kisses* in the context of being immediately preceded by the sequence *the girl*. In other words, any set of context-free rules that generate (correctly) the sequences *the girl kisses the boy* and *the girls kiss the boy* will also generate (incorrectly) the sequences *the girl kiss the boy* and *the girls kisses the boy*. The grammatical phenomenon of Subject-Predicate agreement is sufficient to guarantee the accuracy of: "English is not a context-free language".

Exactly the same phenomenon is used by Emmon Bach in *Syntactic theory* (Bach, 1974, p. 77) to reach the same conclusion. After listing several examples of subject-verb agreement in English main clauses, he says:

> These examples show that to describe the facts of English number agreement is literally impossible using a simple agreement rule of the type given in a phrase-structure grammar, since we cannot guarantee that the noun phrase that determines the agreement will precede (or even be immediately adjacent) to the present-tense verb.

© 2001 by Shuly Wintner

Of course, such phenomena are trivial to describe with context-free grammars, as outlined in section 3.6.

Similarly, Joan Bresnan in *A realistic transformational grammar* (Bresnan, 1978, pp. 37-38) lists some more examples of number agreement in English, indicating that:

> In many cases the number of a verb agrees with that of a noun phrase at some distance from it... this type of syntactic dependency can extend as far as memory or patience permits... the distant type of agreement... cannot be adequately described even by context-sensitive phrase-structure rules, for the possible context is not correctly describable as a finite string of phrases.

It should be emphasized that context-free rules can take care of dependencies that go far beyond "immediate adjacency". For example, we could augment the grammar of example 3.13 to allow relative clause modification of noun phrases by separating singular and plural categories (as suggested in section 3.6) and adding the rules *NPsg* $\rightarrow$ *NPsg that S* and *NPpl* $\rightarrow$ *NPpl that S*, and thus account for sentences in which the subject and the verb agree on number, even though they might be separated from each other by a full clause, whose length might not be limited.

There have been, however, some more serious attempts to demonstrate that natural languages are not context-free. We have seen so far one wrong way of demonstrating that a language is not context-free. How, then does one go about proving such a claim? There are two main techniques. One is direct: there is a lemma, known as the *pumping lemma*, that describes some mathematical property of context-free languages in terms of their string sets. To prove that a particular language is not context-free, one can simply prove that the set of strings of that language does not comply with this property. However, the property is rather complex; direct proofs of non context-freeness are rare.

More useful than the pumping lemma are two closure properties of context-free languages: closure under intersection with regular languages and under homomorphism. Suppose we want to check whether English is context-free. If we can find some regular language, and show that its intersection with English is not context-free, then we know that English itself is not context-free. The target language – in the intersection of English and some regular language – should be one that is obviously not context-free, and usually one uses a variant of what's called "copy languages". The canonical example of a copy language is $\{ww \mid w \in \Sigma^*\}$. In other words, if we can show that when English is intersected with some regular language it yields a copy language, then English is provably not context-free. Using such techniques, it was demonstrated quite convincingly – using data from Dutch, Swiss-German and other languages – that there are certain constructions in natural languages that cannot be generated by context-free grammars.

We will not demonstrate here the non-context-freeness of natural languages, as both the linguistic data and the mathematics of the proof are beyond the scope of this course. It is much simpler, though, to show that natural languages are not regular; in particular, we demonstrate below that English is not a regular language.

A famous example for showing that natural languages are not regular has to do with phenomena known as *center embedding*. The following is a grammatical English sentence:

A white male hired another white male.

The subject, *A white male*, can be modified by the relative clause *whom a white male hired*:

A white male – whom a white male hired – hired another white male.

Now the subject of the relative clause can again be modified by the same relative clause, and in principle there is no bound to the level of embedding allowed by English. It is obtained, then, that the language L_{trg} is a subset of English:

$$L_{trg} = \{\textit{A white male (whom a white male)}^n \textit{ (hired)}^n \textit{ hired another white male} \mid n > 0\}$$

There are two important points in this argument: first, L_{trg} is *not* a regular language. This can be easily shown using a similar argument to the one we used in order to demonstrate that the language $\{a^n b^n \mid n > 0\}$ is not regular, in section 3.1.

© 2001 by Shuly Wintner

The second point is that all the sentences of L_{trg} are grammatical English sentences, while no "similar" sentence, in which the exponents of the clauses *whom a white male* and *hired* differ, is a grammatical English sentence. In other words, L_{trg} is the intersection of the natural language English with the regular set

$$L_{reg} = \{\textit{A white male (whom a white male)}^* \textit{ (hired)}^* \textit{ hired another white male}\}$$

L_{reg} is indeed regular, as it is defined by a regular expression. Now assume towards a contradiction that English were a regular language. Then since the regular languages are closed under intersection, and since L_{reg} is a regular language, then so would have been the intersection of English with L_{reg}, namely L_{trg}. Since L_{trg} is known to be trans-regular, we get at a contradiction, and therefore our assumption is falsified: English is *not* a regular language.

As noted above, similar considerations – albeit with a more complicated set of data, and a more involved mathematical argument – can be used to demonstrate that certain natural languages stand beyond the expressive power of even context-free languages.

4.3 Weak and strong generative capacity

In the discussion of the previous section we only looked at grammars as generating sets of strings (i.e., languages), and ignored the structure that a grammar imposes on the strings in its language. In other words, when we say that English is not a regular language we mean that no regular expression exists whose denotation is the set of all and only the sentences of English. Similarly, when a claim is made that some natural language, say Dutch, is not context-free, it should be read as saying that no context-free grammar exists whose language is Dutch. Such claims are propositions on the *weak generative capacity* of the formalisms involved: the weak generative capacity of regular expressions is insufficient for generating English; the weak generative capacity of context-free languages is insufficient for Dutch. Where natural languages are concerned, however, weak generative capacity might not correctly characterize the relationship between a formalism (such as regular expressions or context-free grammars) and a language (such as English or Dutch). This is because one expects the formalism not only to be able to generate the strings in a language, but also to associate them with "correct" structures.

In the case of context-free grammars, the structure assigned to strings is a derivation tree. Other linguistic formalisms might assign other kinds of objects to their sentences. We say that the *strong generative capacity* of some formalism is sufficient to generate some language if the formalism can (weakly) generate all the strings in the language, and also to assign to them the "correct" structures. Unlike weak generative capacity, which is a properly defined mathematical notion, strong generative capacity is poorly defined, because no clear definition of the "correct" structure for some string in some language exists.

As an extreme example, consider some subset of English, defined with the following grammar, G_1:

$S \rightarrow NP\ VP$	$D \rightarrow the$
$NP \rightarrow D\ N$	$N \rightarrow cat$
	$N \rightarrow hat$
$PP \rightarrow P\ NP$	$P \rightarrow in$
$VP \rightarrow V$	$V \rightarrow sleeps$
$VP \rightarrow V\ NP$	$V \rightarrow saw$
$VP \rightarrow V\ PP$	$V \rightarrow loves$

This grammar defines sentences such as *the cat sleeps, the cat saw the hat, the cat sleeps in the hat*, etc. But G_1 not only generates such sentences; it also associates each of them with a derivation tree. Now observe that the language generated by G_1 is finite: there is no recursion in the rules, and so one can easily enumerate all the sentences in the language.

Now consider an alternative grammar, G_2, with the following rules:

© 2001 by Shuly Wintner

$S \to$ *the cat sleeps*
$S \to$ *the cat sleeps in the cat*
$S \to$ *the cat sleeps in the hat*
$S \to$ *the cat saw the hat*
$S \to$ *the cat loves the hat*
$S \to$ *the cat loves the cat*
...

In other words, G_2 has a production rule for every possible sentence of the language generated by G_1. Since $L(G_1)$ is finite, the number of rules in G_2 is finite, too. Furthermore, by its construction, the language generated by G_2 is identical to the language generated by G_1. In terms of their weak generative capacity, we would say that the grammars are equivalent: they generate exactly the same sets of strings. However, in terms of their strong generative capacity, it is obvious that G_1 and G_2 assign completely different structures to the strings in their (common) language: clearly, the trees admitted by G_1 are more informative than the flat tress admitted by G_2. We might say that G_1 is superior to G_2, as it corresponds more closely to the intuitions we have when we think of the structure of English sentences; but such a claim is not a mathematical proposition and cannot be proven.

G_1 and G_2 above are two context-free grammars, and hence assign the same kind of structure (derivation trees) to the strings in their languages. When two grammars that are instances of *different* formalisms are concerned, matters become more complicated: it is not only difficult to evaluate the "correctness" of the structures assigned to strings; these structures might be completely different entities. For example, in some linguistic theories strings are associated with complex entities called *feature structures* rather than with strings. It is very difficult to define relationships between structures of one formalism and those of another.

To summarize, the picture that emerges from the above discussion is that comparison of different grammars, and therefore also of different linguistic formalisms, can only be done on the basis of their weak generative capacity. This is the notion that grammar equivalence captures; this is the notion that should be used when referring to the appropriateness of some formalism for some language (or a class of languages). Strong generative capacity is interesting in and by itself, but it is not a notion that is amenable to formal, mathematical investigation.

Further reading

The Chomsky Hierarchy of Languages is due to Chomsky (1956). The location of the natural languages in this hierarchy is discussed in a variety of papers, and by far the most readable, enlightening and amusing is Pullum and Gazdar (1982), on which section 4.2 is based. Several other works discussing the non-context-freeness of natural languages are collected in Part III of Savitch et al. (1987). Rounds, Manaster-Ramer, and Friedman (1987) inquire into the relations between formal language theory and linguistic theory, in particular referring to the distinction between weak and strong generative capacity. Works showing that natural languages cannot be described by context-free grammars include Bresnan et al. (1982) (Dutch), Shieber (1985) (Swiss-German) and Manaster-Ramer (1987) (Dutch). Miller (1999) is a recent book dedicated to generative capacity of linguistic formalisms, where strong generative capacity is defined as the model theoretic semantics of a formalism.

© 2001 by Shuly Wintner

Bibliography

Akmajian, Adrian and Frank Heny. 1976. *An Introduction to the Principles of Transformational Syntax*. The MIT Press, Cambridge, Mass.

Bach, Emmon. 1974. *Syntactic Theory*. Holt, Rinehart and Winston, Inc.

Beesley, Kenneth R. and Lauri Karttunen. Forthcoming. *Finite-State Morphology: Xerox Tools and Techniques*.

Bresnan, Joan. 1978. A realistic transformational grammar. In Morris Halle, Joan Bresnan, and George A. Miller, editors, *Linguistic theory and psychological reality*. The MIT Press, Cambridge, Mass, chapter 1, pages 1–59.

Bresnan, Joan, Ronald M. Kaplan, Stanley Peters, and Annie Zaenen. 1982. Cross-serial dependencies in Dutch. *Linguistic Inquiry*, 13(4):613–635.

Chomsky, Noam. 1956. Three models for the description of language. In *I. R. E. transactions on information theory, Proceedings of the symposium on information theory*, volume IT-2, pages 113–123, September.

Chomsky, Noam. 1957. *Syntactic Structures*. Mouton & Co., The Hague, The Netherlands.

Culicover, Peter W. 1976. *Syntax*. Academic Press, New York, NY.

Gazdar, G. E., E. Klein, K. Pullum, and Ivan A. Sag. 1985. *Generalized Phrase Structure Grammar*. Harvard University Press, Cambridge, Mass.

Grinder, John T. and Suzette Haden Elgin. 1973. *Guide to Transformational Grammar*. Holt, Rinehart and Winston, Inc.

Hopcroft, John E. and Jeffrey D. Ullman. 1979. *Introduction to automata theory, languages and computation*. Addison-Wesley Series in Computer Science. Addison-Wesley Publishing Company, Reading, Mass.

Kaplan, Ronald M. and Martin Kay. 1994. Regular models of phonological rule systems. *Computational Linguistics*, 20(3):331–378, September.

Karttunen, Lauri. 1991. Finite-state constraints. In *Proceedings of the International Conference on Current Issues in Computational Linguistics*, Universiti Sains Malaysia, Penang, Malaysia, June. Available from `http://www.xrce.xerox.com/research/mltt/fst/fsrefs.html`.

Karttunen, Lauri, Jean-Pierre Chanod, Gregory Grefenstette, and Anne Schiller. 1996. Regular expressions for language engineering. *Natural Language Engineering*, 2(4):305–328.

Koskenniemi, Kimmo. 1983. *Two-Level Morphology: a general computational model for word-form recognition and production*. The department of general linguistics, University of Helsinki.

Manaster-Ramer, Alexis. 1987. Dutch as a formal language. *Linguistics and Philosophy*, 10:221–246.

Miller, Philip. 1999. *Strong Generative Capacity: The Semantics of Linguistic Formalism*. CSLI Publications, Stanford, California.

Partee, Brabara H., Alice ter Meulen, and Robert E. Wall. 1990. *Mathematical Methods in Linguistics*, volume 30 of *Studies in Linguistics and Philosophy*. Kluwer Academic Publishers, Dordrecht.

Pullum, Geoffrey K. and Gerald Gazdar. 1982. Natural languages and context-free languages. *Linguistics and Philosophy*, 4:471–504.

Roche, Emmanuel and Yves Schabes, editors. 1997a. *Finite-State Language Processing*. Language, Speech and Communication. MIT Press, Cambridge, MA.

Roche, Emmanuel and Yves Schabes. 1997b. Introduction. In Emmanuel Roche and Yves Schabes, editors, *Finite-State Language Processing*, Language, Speech and Communication. MIT Press, Cambridge, MA, chapter 1, pages 1–65.

Rounds, William C., Alexis Manaster-Ramer, and Joyce Friedman. 1987. Finding natural languages a home in formal language theory. In Alexis Manaster-Ramer, editor, *Mathematics of Language*. John Benjamins Publishing Company, Amsterdam/Philadelphia, pages 349–359.

Savitch, Walter J., Emmon Bach, William Marsh, and Gila Safran-Naveh, editors. 1987. *The formal complexity of natural language*, volume 33 of *Studies in Linguistics and Philosophy*. D. Reidel, Dordrecht.

Shieber, Stuart M. 1985. Evidence against the context-freeness of natural language. *Linguistics and Philosophy*, 8:333–343.

© 2001 by Shuly Wintner