Chapter 2

Formal language theory

We need a certain amount of formal language theory to understand the concepts to be treated in subsequent chapters. These may be familiar notions to students of computer science, mathematics, or formal linguistic theory. Here are the notions I want to make sure we understand. I will only treat these in sufficient depth to understand relevant notions in statistical NLP; there is much more to say about each.

Formal Language A simple mathematical characterization of what a language is.

Regular Language A fairly simple kind of formal language.

Finite State Machines A very simple model of a computer.

Context-free language A somewhat more complex kind of formal language.

Re-write grammars A characterization of a language in terms of a set of rules converting some set of symbols to surface strings.

Let's consider each in turn.

2.1 Formal language

We will take a formal language to be a possibly infinite set of words constructed from some finite alphabet. For example, we might have an alphabet

$\{a, b, c\}$. One language over this alphabet is every possible string containing an even number of the symbol a. Another language would be every string beginning with c. There are an infinite number of possible languages that can be defined over such a vocabulary, and these languages themselves can be infinite in size. The way we specify these languages determines what family of languages the language belongs to.

2.2 Regular Language

A *regular* language is a restricted kind of formal language, yet given any finite alphabet there are an infinite number of possible regular languages and regular languages themselves can of course be infinite.

A regular language is a language that can be described using only three operations: *union*, *concatenation*, and *Kleene star*. Union means "or". For example, given the alphabet $\{a, b\}$, one could define the regular language $a(a|b)$. This regular language is a finite one and includes only the strings: $\{aa, ab\}$. Concatenation refers to whether one can stipulate the ordering of elements. For example, the regular language $a(a|b)$ stipulates that the symbol a precedes the union of $(a|b)$. Union and concatenation can proceed at different "levels" too: $(aa|b)$ designates a different (finite) regular language: $\{aa, b\}$. Finally there is Kleene star which designates an unbounded number of the preceeding element. For example, $a(b|c)a^*$ designates the infinite regular language wherein every word begins with ab or ac and then some number of a's, e.g. $\{ab, ac, aba, aca, abaa, acaa, \ldots\}$.

If a language can be described with a *regular expression*, then it falls within the class of regular languages. What is an example of a language that is not regular? The language $a_n b_n$, ab, $aabb$, $aaabbb$, etc., is defined such that any word has n instances of a followed by n instances of b. This language cannot be described with concatenation, union, and Kleene star.

Let's digress briefly to consider why the set of regular languages might be interesting linguistically. It's been argued that phonology and morphology can perhaps be described in terms of regular languages.[1] (Syntax in its entirety clearly cannot.) For example, imagine we have the following simple generalization with respect to syllabification. In some language L, words must be parsed into syllables. Moreover, these syllables are of the form CV

[1] This point was made with respect to phonology by Kaplan & Kay (1994) and with respect to morphology by Karttunen (1983).

or CVC. This can be expressed using the following regular expression to describe words: (CV|CVC)(CV|CVC)*.[2] We can also express more traditional phonological generalizations, but I leave this as an exercise for the reader...

2.3 Finite State Machines

It turns out that all and only the set of languages that we can describe with regular expressions can be *accepted* or *generated* with a particularly constrained computational model: the finite state machine or automaton (FSM or FSA). What this means intuitively is that if some language or computational problem can be characterized in terms of a regular language or a regular relation, then it is a problem that in at least some ways can be treated with minimal computational power.

What is a finite state machine? A finite state machine can be thought of as a set of *states* linked by *arcs*. This machine is associated with a *tape* on which are written symbols. The tape is fed in at one end of the machine and the symbols are read off one by one. (The tape proceeds in only one direction.) At any one point in time, the machine is "in" one of its states. As each symbol is read, the machine moves to a new state. If, after the whole tape is read, the machine is in one of a specific subset of its states, a "final" state, then the machine is said to "halt" on that input; that input is accepted. If, on the other hand, after the tape is read, the machine is not in a specified accepting/final state, then the input is not accepted.

A finite state machine can be characterized more formally in terms of four things: a finite set of states, a single start state, some number of final states, and a set of rules that indicate what state changes are possible with what symbol. Each rule indicates what symbol allows the machine to move from some state to some other state.

Consider the following example (figure 2.1). Here we define the alphabet of symbols as $\{a, b\}$ and the states as $\{s_1, s_2, s_3\}$. We designate s_1 as the starting state and s_3 as the final state. The former is indicated with a small arrow and the latter with a double circle. There is an arc from s_1 to s_2 labeled a indicating that the first symbol read can be an a. There is another arc labelled b indicating that the first symbol can also be a b. There is a loop from s_2 to s_2 labeled a, indicating that the first symbol can be followed by

[2]This can be expressed more succinctly if we make use of the empty symbol: ϵ. For example: CV(C|ϵ)(CV(C|ϵ))*.

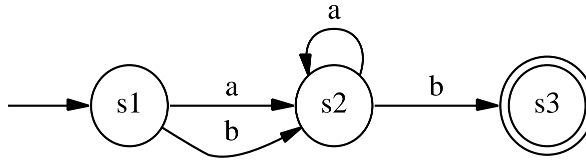

Figure 2.1: A simple FSA

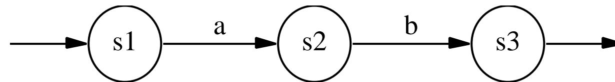

Figure 2.2: Concatenation

any number of instances of a. Finally, there is an arc from s_2 to s_3 labeled b, indicating that the string must terminate with a b. In this case, the language accepted by this FSA is: $(a|b)a^*b$. In this language every word is composed of a single a or b followed by any number of instances of a, followed by a single b.

In fact, one can show that FSAs can describe all and only the set of regular languages. This can be proven formally, but let's just look at the basic logic here. We have defined regular languages as those describable in terms of concatenation, union, and Kleene star. These three operations are mirrored by three properties of FSAs. First, concatenation is mirrored by the sequencing of arcs and states. To describe the fact that a symbol a must precede a symbol b in some regular language, e.g. $\ldots ab \ldots$, one specifies a sequence of arcs and states as in figure 2.2.

To describe the fact that there is a choice between two symbols or strings, e.g. $\ldots (a|b) \ldots$, one has multiple arcs leaving the same state, as in figure 2.3.

Finally, to capture Kleene star, e.g. $\ldots a^* \ldots$, one posits a loop (figure 2.4).

The regular languages are "closed" under the same three operations that define them. In other words, if you apply union, concatenation or Kleene star to some number of regular languages, you still end up with a regular language.

Of course, FSAs are likewise "closed". For example, concatenating two regular languages L_1 and L_2, e.g. $L_1 L_2$ produces a new regular language. The

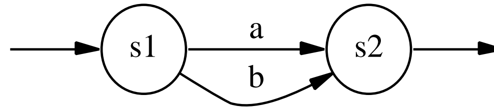

Figure 2.3: Union

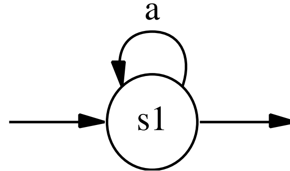

Figure 2.4: Kleene star

union of two regular languages is also a regular language, e.g. $(L_1|L_2)$. Finally, applying Kleene star to a regular language produces a regular language: $(L_1)^*$.

The same three operations applied to FSAs produce new FSAs. First, two FSAs can be concatenated, producing a new FSA. Consider the two FSAs in figure 2.5.

These can be concatenated (by merging nodes S_2 and S_3) to produce the new FSA in figure 2.6.

We can also create the union of these two FSAs (by merging S_1 and S_3). See figure 2.7.

Finally, we can perform the equivalent of Kleene star on an FSA by inserting a loop from the final state back through to the final state again. For example, the FSA in figure 2.8 makes a loop of the first of the two preceding FSAs.

FSAs and regular languages are also closed under intersection and complementation.[3]

FSAs come in two basic types: *deterministic* and *non-deterministic*. A deterministic automaton is one where every state has one and only one out-

[3]Do I want to show how this works?

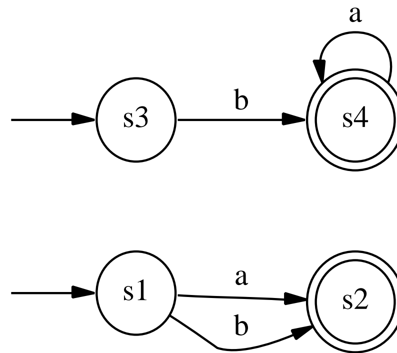

Figure 2.5: Two FSAs

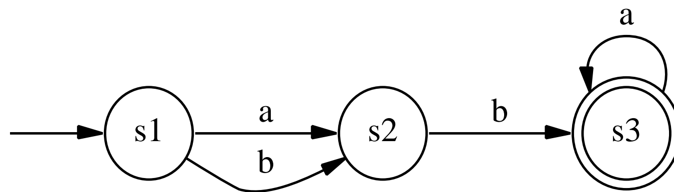

Figure 2.6: Concatenated FSAs

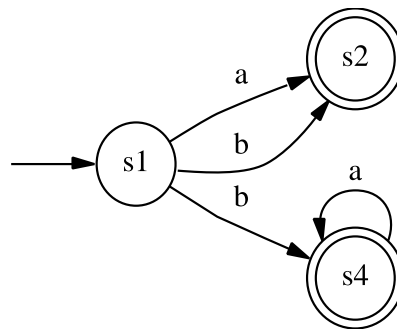

Figure 2.7: union of two FSAs

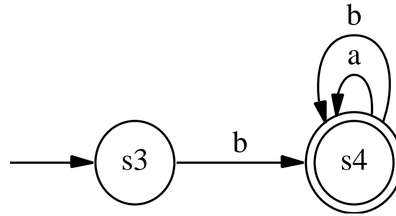

Figure 2.8: Kleene star

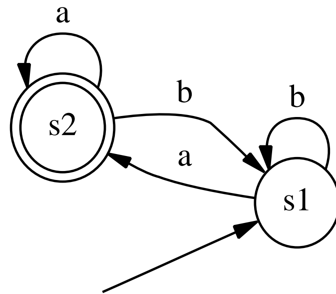

Figure 2.9: Deterministic FSA

going arc for each symbol in the alphabet. For example, the FSA in figure 2.9 is deterministic with respect to the alphabet $\{a, b\}$. The automaton in figure 2.10 is not, however, deterministic with respect to $\{a, b\}$.

It is possible for non-deterministic FSAs to contain arcs with no symbol; such arcs are termed "null transitions".[4] Any number of null transitions can be followed at any point. See, for example, figure 2.11.

One can show formally that non-deterministic FSAs are just as powerful as deterministic FSAs. That is the set of languages that can be generated with a deterministic FSA is exactly the same as the set of languages that can be generated with a non-deterministic FSA.

Moreover, null transitions do not add to the power of a FSA. FSAs with null transitions can produce exactly the same set of (regular) languages as

[4] Sometimes such arcs are labeled with ϵ.

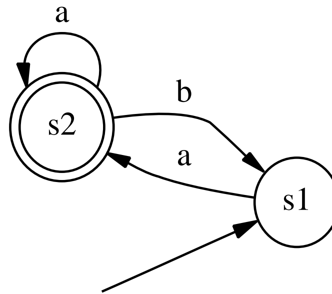

Figure 2.10: Non-deterministic FSA

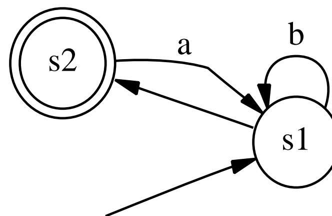

Figure 2.11: FSA with null transition

terminals:	a, b
non-terminals:	A, B
starting node:	A
production rules:	$A \rightarrow a\ B\ b$
	$B \rightarrow a\ b$
	$B \rightarrow \emptyset$

Figure 2.12: context-free grammar

produced by a FSA with no null transitions.[5]

2.4 Finite State Transducer

The regular languages and FSAs are mirrored by the set of "regular relations", and Finite State Transducers (FSTs). FSTs are just like FSAs, except that each arc is labeled with two symbols. Following the machine metaphor, a FST can be thought of as a machine with two tapes. The tapes are read simultaneously and the machine thus defines a mapping across the two tapes. Such models have been widely used to model phonological and morphological systems, but I'll have no more to say about them here.[6]

2.5 Context-free languages

There is another family of formal languages, slightly more complex than the regular languages: *context-free* (CF) languages. These can be characterized as i) a set of terminal elements, ii) a set of non-terminal elements, iii) one or more non-terminals designated as *starting* elements, and iv) a set of *production rules* mapping non-terminals to strings of terminals and non-terminals. See, for example, table 2.12.

The system in 2.12 demonstrates that languages defined as CF can be more powerful than regular languages since the system above describes the $a_n b_n$ language, which we already said was not regular.

[5] Do I want to show this?

[6] See Kaplan & Kay (1994) and Karttunen (1983).

2.6 Finite State Machine with memory

It turns out that the context-free languages can be described with an extension of the finite-state idea. Basically, if a FSM is augmented with a memory, then it can describe a CF language.

The particular memory that is required is a *push-down* memory. The basic idea is that symbols can be recorded in memory one by one. They can then be recalled from memory in reverse order. For example, if elements a_1, a_2, and a_3 are recorded in the push-down memory in that order, then can be recalled in the opposite order: a_3, a_2, a_1.

Describing the $a_n b_n$ language with such a system is then a straightforward matter of placing some symbol in the stack each time the symbol a is read from the tape. The symbols are then read off the stack and a b is read off the tape. If, at the end of the string, the stack is empty, then the string is accepted.

2.7 Re-write rules

We have shown that a context-free language can be described using a re-write grammar. There are interesting restrictions that can be put on re-write grammars that can result in a more restrictive language type. We consider two of these here.

2.7.1 Chomsky-Normal Grammars

One interesting restriction is called *Chomsky Normal Form* (CNF). Re-write grammars in this form are no more restrictive than CF grammars, but CNF is often used in proofs regarding the CF languages.

A re-write grammar in CNF exhibits only two kinds of re-write rules. First, there are rules of the form $A \rightarrow BC$, where A, B, and C are non-terminal symbols. The other allowed rule type is $A \rightarrow a$, where A is a non-terminal and a is a single terminal.

This is obviously an instance of a re-write grammar, but we can show that any rewrite grammar can be converted to this form. We do this by showing how rules that do not fit these forms can be converted into rules confirming to the CNF schema.

First, for any rule of the form $A \rightarrow a_1 \ldots a_n$, we create new non-terminal elements $X_1 \ldots X_n$, we add new productions $X_1 \rightarrow a_1$ etc., and we replace the original rule with a rule $A \rightarrow X_1 \ldots X_n$.

Second, any rule of the form $A \rightarrow aBb$ is handled in a similar manner, except that any non-terminals on the righthand side are left as is.

Third, any rule of the form $A \rightarrow B$ is eliminated. We first find any rules of the form $B \rightarrow \ldots$. We then add rules to the grammar of the form $A \rightarrow \ldots$.

Finally, after we have (recursively) made the changes above, the only rules left violating the CNF schema will be those of the form $A \rightarrow B_1 \ldots B_n$, where $n > 2$. These are eliminated by adding new non-terminals $X_1 \ldots X_n$ and replacing the offending rule with the following rules: $A \rightarrow B_1X_1$, $X_1 \rightarrow B_2X_2$, etc.

Since grammars in CNF are instances of CF grammars and since, as we have just shown, any CF grammar can be converted into CNF, it follows that CNF is equivalent to CF.

2.7.2 Linear grammars

We now consider the class of right-linear and left-linear re-write grammars. It will turn out that the restriction to linear form amounts to a significant restriction in the power of a re-write grammar, such that these grammars are equivalent to the regular languages. Thus these two classes of grammars are also referred to as the *regular* grammars.

A right-linear grammar has rules of only two forms. First, we have $A \rightarrow a_1 \ldots a_n$, where a is a terminal element. Second, we have rules of the form $A \rightarrow a_1 \ldots a_nB$, where B is a single non-terminal element.

To show that a right-linear grammar is equivalent to a regular language we show how any right-linear grammar can be expressed with a FSA.[7]

First, for any rule of the form $A \rightarrow a_1 \ldots a_n$, we posit a series of states with each of $a_1 \ldots a_n$ labelling the arcs that connect them, beginning with a state we will label A (concatenation).

Second, if there are multiple rules with some non-terminal A on the left, then there will be multiple arcs from A in the FSA (union).

Third, for any rule of the form $A \rightarrow a_1 \ldots a_nB$, we follow the first rule above, terminating the string of states with a state labeled B.

[7] This can be proved rigorously, but we simply sketch the proof informally here.

Finally, if there are any "cycles" among the rules, e.g. $A \to a_1 \ldots a_n A$ or sets of the form $A \to a_1 \ldots a_n B$ and $B \to b_1 \ldots b_n A$, then there will be analogous loops in the FSA (Kleene star).

Since any right-linear grammar can be expressed as a FSA, it follows that languages defined in these terms are regular.

There are also *left*-linear grammars where the allowed rule types are $A \to a_1 \ldots a_n$ and $A \to a_1 \ldots a_n B$. These are also equivalent to the regular languages. The proof of this is left to the reader.