Chapter 9

Formal language theory

We have now seen methods for learning to label individual words, vectors of word counts, and sequences of words; we will soon proceed to more complex structural transformations. Most of these techniques could apply to counts or sequences from any discrete vocabulary; there is nothing fundamentally linguistic about, say, a hidden Markov model. This raises a basic question that this text has not yet considered: what is a language?

This chapter will take the perspective of **formal language theory**, in which a language is defined as a set of **strings**, each of which is a sequence of elements from a finite alphabet. For interesting languages, there are an infinite number of strings that are in the language, and an infinite number of strings that are not. For example:

- the set of all even-length sequences from the alphabet $\{a, b\}$, e.g., $\{\varnothing, aa, ab, ba, bb, aaaa, aaab, \ldots\}$;
- the set of all sequences from the alphabet $\{a, b\}$ that contain aaa as a substring, e.g., $\{aaa, aaaa, baaa, aaab, \ldots\}$;
- the set of all sequences of English words (drawn from a finite dictionary) that contain at least one verb (a finite subset of the dictionary);
- the PYTHON programming language.

Formal language theory defines classes of languages and their computational properties. Of particular interest is the computational complexity of solving the **membership problem** — determining whether a string is in a language. The chapter will focus on three classes of formal languages: regular, context-free, and "mildly" context-sensitive languages.

A key insight of 20th century linguistics is that formal language theory can be usefully applied to natural languages such as English, by designing formal languages that capture as many properties of the natural language as possible. For many such formalisms, a useful linguistic analysis comes as a byproduct of solving the membership problem. The membership problem can be generalized to the problems of *scoring* strings for their acceptability (as in language modeling), and of **transducing** one string into another (as in translation).

9.1 Regular languages

If you have written a **regular expression**, then you have defined a **regular language**: a regular language is any language that can be defined by a regular expression. Formally, a regular expression can include the following elements:

- A **literal character** drawn from some finite alphabet Σ.
- The **empty string** ϵ.
- The concatenation of two regular expressions RS, where R and S are both regular expressions. The resulting expression accepts any string that can be decomposed $x = yz$, where y is accepted by R and z is accepted by S.
- The alternation $R \mid S$, where R and S are both regular expressions. The resulting expression accepts a string x if it is accepted by R or it is accepted by S.
- The **Kleene star** R^*, which accepts any string x that can be decomposed into a sequence of strings which are all accepted by R.
- Parenthesization (R), which is used to limit the scope of the concatenation, alternation, and Kleene star operators.

Here are some example regular expressions:

- The set of all even length strings on the alphabet $\{a, b\}$: $((aa)|(ab)|(ba)|(bb))^*$
- The set of all sequences of the alphabet $\{a, b\}$ that contain aaa as a substring: $(a|b)^*aaa(a|b)^*$
- The set of all sequences of English words that contain at least one verb: W^*VW^*, where W is an alternation between all words in the dictionary, and V is an alternation between all verbs ($V \subseteq W$).

This list does not include a regular expression for the Python programming language, because this language is not regular — there is no regular expression that can capture its syntax. We will discuss why towards the end of this section.

Regular languages are **closed** under union, intersection, and concatenation. This means that if two languages L_1 and L_2 are regular, then so are the languages $L_1 \cup L_2$, $L_1 \cap L_2$, and the language of strings that can be decomposed as $s = tu$, with $s \in L_1$ and $t \in L_2$. Regular languages are also closed under negation: if L is regular, then so is the language $\overline{L} = \{s \notin L\}$.

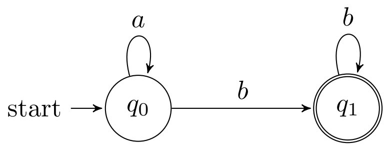


Figure 9.1: State diagram for the finite state acceptor M_1.

9.1.1 Finite state acceptors

A regular expression defines a regular language, but does not give an algorithm for determining whether a string is in the language that it defines. **Finite state automata** are theoretical models of computation on regular languages, which involve transitions between a finite number of states. The most basic type of finite state automaton is the **finite state acceptor (FSA)**, which describes the computation involved in testing if a string is a member of a language. Formally, a finite state acceptor is a tuple $M = (Q, \Sigma, q_0, F, \delta)$, consisting of:

- a finite alphabet Σ of input symbols;
- a finite set of states $Q = \{q_0, q_1, \ldots, q_n\}$;
- a start state $q_0 \in Q$;
- a set of final states $F \subseteq Q$;
- a transition function $\delta : Q \times (\Sigma \cup \{\epsilon\}) \to 2^Q$. The transition function maps from a state and an input symbol (or empty string ϵ) to a *set* of possible resulting states.

A **path** in M is a sequence of transitions, $\pi = t_1, t_2, \ldots, t_N$, where each t_i traverses an arc in the transition function δ. The finite state acceptor M accepts a string ω if there is an accepting path, in which the initial transition t_1 begins at the start state q_0, the final transition t_N terminates in a final state in Q, and the entire input ω is consumed.

Example

Consider the following FSA, M_1.

$$\Sigma =\{a, b\} \tag{9.1}$$
$$Q =\{q_0, q_1\} \tag{9.2}$$
$$F =\{q_1\} \tag{9.3}$$
$$\delta =\{(q_0, a) \to q_0,\ (q_0, b) \to q_1,\ (q_1, b) \to q_1\}. \tag{9.4}$$

This FSA defines a language over an alphabet of two symbols, *a* and *b*. The transition function δ is written as a set of arcs: $(q_0, a) \to q_0$ says that if the machine is in state

q_0 and reads symbol a, it stays in q_0. Figure 9.1 provides a graphical representation of M_1. Because each pair of initial state and symbol has at most one resulting state, M_1 is **deterministic**: each string ω induces at most one accepting path. Note that there are no transitions for the symbol a in state q_1; if a is encountered in q_1, then the acceptor is stuck, and the input string is rejected.

What strings does M_1 accept? The start state is q_0, and we have to get to q_1, since this is the only final state. Any number of a symbols can be consumed in q_0, but a b symbol is required to transition to q_1. Once there, any number of b symbols can be consumed, but an a symbol cannot. So the regular expression corresponding to the language defined by M_1 is a^*bb^*.

Computational properties of finite state acceptors

The key computational question for finite state acceptors is: how fast can we determine whether a string is accepted? For determistic FSAs, this computation can be performed by Dijkstra's algorithm, with time complexity $\mathcal{O}(V \log V + E)$, where V is the number of vertices in the FSA, and E is the number of edges (Cormen et al., 2009). Non-deterministic FSAs (NFSAs) can include multiple transitions from a given symbol and state. Any NSFA can be converted into a deterministic FSA, but the resulting automaton may have a number of states that is exponential in the number of size of the original NFSA (Mohri et al., 2002).

9.1.2 Morphology as a regular language

Many words have internal structure, such as prefixes and suffixes that shape their meaning. The study of word-internal structure is the domain of **morphology**, of which there are two main types:

- **Derivational morphology** describes the use of affixes to convert a word from one grammatical category to another (e.g., from the noun *grace* to the adjective *graceful*), or to change the meaning of the word (e.g., from *grace* to *disgrace*).
- **Inflectional morphology** describes the addition of details such as gender, number, person, and tense (e.g., the *-ed* suffix for past tense in English).

Morphology is a rich topic in linguistics, deserving of a course in its own right.[1] The focus here will be on the use of finite state automata for morphological analysis. The

[1]A good starting point would be a chapter from a linguistics textbook (e.g., Akmajian et al., 2010; Bender, 2013). A key simplification in this chapter is the focus on affixes at the sole method of derivation and inflection. English makes use of affixes, but also incorporates **apophony**, such as the inflection of *foot* to *feet*. Semitic languages like Arabic and Hebrew feature a template-based system of morphology, in which roots are triples of consonants (e.g., *ktb*), and words are created by adding vowels: ***k****a****t****a****b****a* (Arabic: he wrote), ***k****u****t****u****b*** (books), *ma****ktab*** (desk). For more detail on morphology, see texts from Haspelmath and Sims (2013) and Lieber (2015).

current section deals with derivational morphology; inflectional morphology is discussed in § 9.1.4.

Suppose that we want to write a program that accepts only those words that are constructed in accordance with the rules of English derivational morphology:

(9.1) a. grace, graceful, gracefully, *gracelyful

b. disgrace, *ungrace, disgraceful, disgracefully

c. allure, *allureful, alluring, alluringly

d. fairness, unfair, *disfair, fairly

(Recall that the asterisk indicates that a linguistic example is judged unacceptable by fluent speakers of a language.) These examples cover only a tiny corner of English derivational morphology, but a number of things stand out. The suffix *-ful* converts the nouns *grace* and *disgrace* into adjectives, and the suffix *-ly* converts adjectives into adverbs. These suffixes must be applied in the correct order, as shown by the unacceptability of **gracelyful*. The *-ful* suffix works for only some words, as shown by the use of *alluring* as the adjectival form of *allure*. Other changes are made with prefixes, such as the derivation of *disgrace* from *grace*, which roughly corresponds to a negation; however, *fair* is negated with the *un-* prefix instead. Finally, while the first three examples suggest that the direction of derivation is noun → adjective → adverb, the example of *fair* suggests that the adjective can also be the base form, with the *-ness* suffix performing the conversion to a noun.

Can we build a computer program that accepts only well-formed English words, and rejects all others? This might at first seem trivial to solve with a brute-force attack: simply make a dictionary of all valid English words. But such an approach fails to account for morphological **productivity** — the applicability of existing morphological rules to new words and names, such as *Trump* to *Trumpy* and *Trumpkin*, and *Clinton* to *Clintonian* and *Clintonite*. We need an approach that represents morphological rules explicitly, and for this we will try a finite state acceptor.

The dictionary approach can be implemented as a finite state acceptor, with the vocabulary Σ equal to the vocabulary of English, and a transition from the start state to the accepting state for each word. But this would of course fail to generalize beyond the original vocabulary, and would not capture anything about the **morphotactic** rules that govern derivations from new words. The first step towards a more general approach is shown in Figure 9.2, which is the state diagram for a finite state acceptor in which the vocabulary consists of **morphemes**, which include **stems** (e.g., *grace, allure*) and **affixes** (e.g., *dis-, -ing, -ly*). This finite state acceptor consists of a set of paths leading away from the start state, with derivational affixes added along the path. Except for q_{neg}, the states on these paths are all final, so the FSA will accept *disgrace*, *disgraceful*, and *disgracefully*, but not *dis-*.

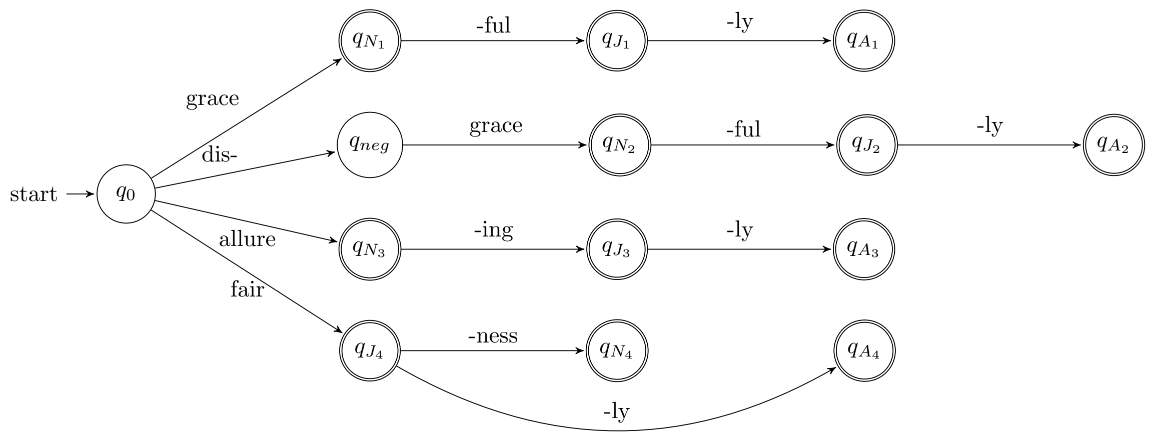


Figure 9.2: A finite state acceptor for a fragment of English derivational morphology. Each path represents possible derivations from a single root form.

This FSA can be **minimized** to the form shown in Figure 9.3, which makes the generality of the finite state approach more apparent. For example, the transition from q_0 to q_{J_2} can be made to accept not only *fair* but any single-morpheme (**monomorphemic**) adjective that takes *-ness* and *-ly* as suffixes. In this way, the finite state acceptor can easily be extended: as new word stems are added to the vocabulary, their derived forms will be accepted automatically. Of course, this FSA would still need to be extended considerably to cover even this small fragment of English morphology. As shown by cases like *music* → *musical*, *athlete* → *athletic*, English includes several classes of nouns, each with its own rules for derivation.

The FSAs shown in Figure 9.2 and 9.3 accept *allureing*, not *alluring*. This reflects a distinction between morphology — the question of which morphemes to use, and in what order — and **orthography** — the question of how the morphemes are rendered in written language. Just as orthography requires dropping the *e* preceding the *-ing* suffix, **phonology** imposes a related set of constraints on how words are rendered in speech. As we will see soon, these issues can be handled by **finite state!transducers**, which are finite state automata that take inputs and produce outputs.

9.1.3 Weighted finite state acceptors

According to the FSA treatment of morphology, every word is either in or out of the language, with no wiggle room. Perhaps you agree that *musicky* and *fishful* are not valid English words; but if forced to choose, you probably find *a fishful stew* or *a musicky tribute* preferable to *behaving disgracelyful*. Rather than asking whether a word is acceptable, we might like to ask how acceptable it is. Aronoff (1976, page 36) puts it another way:

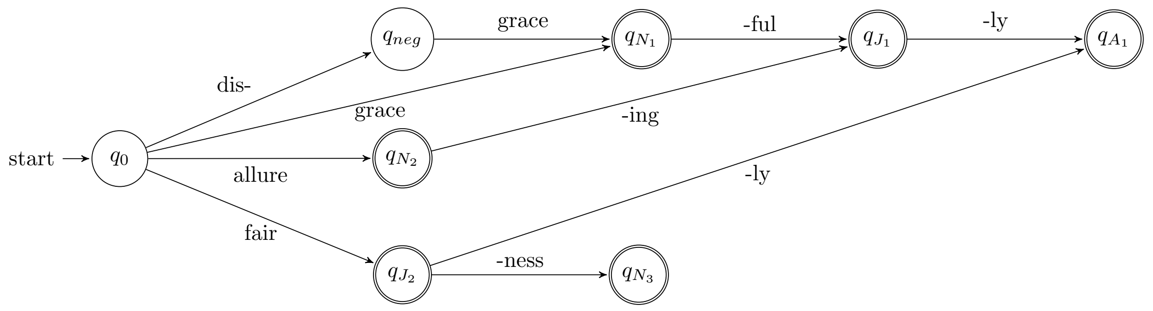


Figure 9.3: Minimization of the finite state acceptor shown in Figure 9.2.

"Though many things are possible in morphology, some are more possible than others." But finite state acceptors give no way to express preferences among technically valid choices.

Weighted finite state acceptors (WFSAs) are generalizations of FSAs, in which each accepting path is assigned a score, computed from the transitions, the initial state, and the final state. Formally, a weighted finite state acceptor $M = (Q, \Sigma, \lambda, \rho, \delta)$ consists of:

- a finite set of states $Q = \{q_0, q_1, \ldots, q_n\}$;
- a finite alphabet Σ of input symbols;
- an initial weight function, $\lambda : Q \to \mathbb{R}$;
- a final weight function $\rho : Q \to \mathbb{R}$;
- a transition function $\delta : Q \times \Sigma \times Q \to \mathbb{R}$.

WFSAs depart from the FSA formalism in three ways: every state can be an initial state, with score $\lambda(q)$; every state can be an accepting state, with score $\rho(q)$; transitions are possible between any pair of states on any input, with a score $\delta(q_i, \omega, q_j)$. Nonetheless, FSAs can be viewed as a special case: for any FSA M we can build an equivalent WFSA by setting $\lambda(q) = \infty$ for all $q \neq q_0$, $\rho(q) = \infty$ for all $q \notin F$, and $\delta(q_i, \omega, q_j) = \infty$ for all transitions $\{(q_1, \omega) \to q_2\}$ that are not permitted by the transition function of M.

The total score for any path $\pi = t_1, t_2, \ldots, t_N$ is equal to the sum of these scores,

$$d(\pi) = \lambda(\text{from-state}(t_1)) + \sum_{n}^{N} \delta(t_n) + \rho(\text{to-state}(t_N)). \qquad [9.5]$$

A **shortest-path algorithm** is used to find the minimum-cost path through a WFSA for string ω, with time complexity $\mathcal{O}(E + V \log V)$, where E is the number of edges and V is the number of vertices (Cormen et al., 2009).[2]

[2]Shortest-path algorithms find the path with the minimum cost. In many cases, the path weights are log

N-gram language models as WFSAs

In n-**gram language models** (see § 6.1), the probability of a sequence of tokens $w_1, w_2, \ldots, w_M$ is modeled as,

$$\mathrm{p}(w_1, \ldots, w_M) \approx \prod_{m=1}^{M} \mathrm{p}_n(w_m \mid w_{m-1}, \ldots, w_{m-n+1}). \qquad [9.6]$$

The log probability under an n-gram language model can be modeled in a WFSA. First consider a unigram language model. We need only a single state q_0, with transition scores $\delta(q_0, \omega, q_0) = \log \mathrm{p}_1(\omega)$. The initial and final scores can be set to zero. Then the path score for $w_1, w_2, \ldots, w_M$ is equal to,

$$0 + \sum_{m}^{M} \delta(q_0, w_m, q_0) + 0 = \sum_{m}^{M} \log \mathrm{p}_1(w_m). \qquad [9.7]$$

For an n-gram language model with $n > 1$, we need probabilities that condition on the past history. For example, in a bigram language model, the transition weights must represent $\log \mathrm{p}_2(w_m \mid w_{m-1})$. The transition scoring function must somehow "remember" the previous word or words. This can be done by adding more states: to model the bigram probability $\mathrm{p}_2(w_m \mid w_{m-1})$, we need a state for every possible w_{m-1} — a total of V states. The construction indexes each state q_i by a context event $w_{m-1} = i$. The weights are then assigned as follows:

$$\begin{aligned}
\delta(q_i, \omega, q_j) &= \begin{cases} \log \Pr(w_m = j \mid w_{m-1} = i), & \omega = j \\ -\infty, & \omega \neq j \end{cases} \\
\lambda(q_i) &= \log \Pr(w_1 = i \mid w_0 = \square) \\
\rho(q_i) &= \log \Pr(w_{M+1} = \blacksquare \mid w_M = i).
\end{aligned}$$

The transition function is designed to ensure that the context is recorded accurately: we can move to state j on input ω only if $\omega = j$; otherwise, transitioning to state j is forbidden by the weight of $-\infty$. The initial weight function $\lambda(q_i)$ is the log probability of receiving i as the first token, and the final weight function $\rho(q_i)$ is the log probability of receiving an "end-of-string" token after observing $w_M = i$.

*Semiring weighted finite state acceptors

The n-gram language model WFSA is deterministic: each input has exactly one accepting path, for which the WFSA computes a score. In non-deterministic WFSAs, a given input

probabilities, so we want the path with the maximum score, which can be accomplished by making each local score into a *negative* log-probability.

may have multiple accepting paths. In some applications, the score for the input is aggregated across all such paths. Such aggregate scores can be computed by generalizing WFSAs with **semiring notation**, first introduced in § 7.7.3.

Let $d(\pi)$ represent the total score for path $\pi = t_1, t_2, \ldots, t_N$, which is computed as,

$$d(\pi) = \lambda(\text{from-state}(t_1)) \otimes \delta(t_1) \otimes \delta(t_2) \otimes \ldots \otimes \delta(t_N) \otimes \rho(\text{to-state}(t_N)). \qquad [9.8]$$

This is a generalization of Equation 9.5 to semiring notation, using the semiring multiplication operator $\otimes$ in place of addition.

Now let $s(\omega)$ represent the total score for all paths $\Pi(\omega)$ that consume input ω,

$$s(\omega) = \bigoplus_{\pi \in \Pi(\omega)} d(\pi). \qquad [9.9]$$

Here, semiring addition ($\oplus$) is used to combine the scores of multiple paths.

The generalization to semirings covers a number of useful special cases. In the log-probability semiring, multiplication is defined as $\log \mathrm{p}(x) \otimes \log \mathrm{p}(y) = \log \mathrm{p}(x) + \log \mathrm{p}(y)$, and addition is defined as $\log \mathrm{p}(x) \oplus \log \mathrm{p}(y) = \log(\mathrm{p}(x) + \mathrm{p}(y))$. Thus, $s(\omega)$ represents the log-probability of accepting input ω, marginalizing over all paths $\pi \in \Pi(\omega)$. In the **boolean semiring**, the $\otimes$ operator is logical conjunction, and the $\oplus$ operator is logical disjunction. This reduces to the special case of unweighted finite state acceptors, where the score $s(\omega)$ is a boolean indicating whether there exists any accepting path for ω. In the **tropical semiring**, the $\oplus$ operator is a maximum, so the resulting score is the score of the best-scoring path through the WFSA. The OPENFST toolkit uses semirings and polymorphism to implement general algorithms for weighted finite state automata (Allauzen et al., 2007).

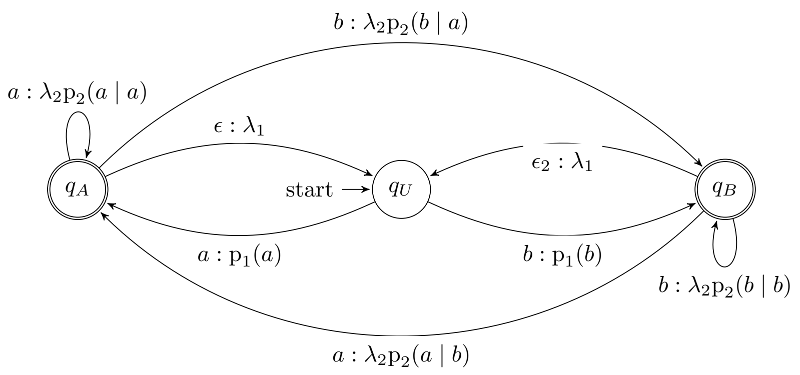
*Interpolated n-gram language models

Recall from § 6.2.3 that an **interpolated n-gram language model** combines the probabilities from multiple n-gram models. For example, an interpolated bigram language model computes the probability,

$$\hat{\mathrm{p}}(w_m \mid w_{m-1}) = \lambda_1 \mathrm{p}_1(w_m) + \lambda_2 \mathrm{p}_2(w_m \mid w_{m-1}), \qquad [9.10]$$

with $\hat{\mathrm{p}}$ indicating the interpolated probability, p_2 indicating the bigram probability, and p_1 indicating the unigram probability. Setting $\lambda_2 = (1 - \lambda_1)$ ensures that the probabilities sum to one.

Interpolated bigram language models can be implemented using a non-deterministic WFSA (Knight and May, 2009). The basic idea is shown in Figure 9.4. In an interpolated bigram language model, there is one state for each element in the vocabulary — in this



$b : \lambda_2 p_2(b \mid a)$
$a : \lambda_2 p_2(a \mid a)$
$\epsilon : \lambda_1$
$\epsilon_2 : \lambda_1$
q_A
start
q_U
q_B
$a : p_1(a)$
$b : p_1(b)$
$b : \lambda_2 p_2(b \mid b)$
$a : \lambda_2 p_2(a \mid b)$

Figure 9.4: WFSA implementing an interpolated bigram/unigram language model, on the alphabet $\Sigma = \{a, b\}$. For simplicity, the WFSA is contrained to force the first token to be generated from the unigram model, and does not model the emission of the end-of-sequence token.

case, the states q_A and q_B — which are capture the contextual conditioning in the bigram probabilities. To model unigram probabilities, there is an additional state q_U, which "forgets" the context. Transitions out of q_U involve unigram probabilities, $p_1(a)$ and $p_2(b)$; transitions into q_U emit the empty symbol ϵ, and have probability λ_1, reflecting the interpolation weight for the unigram model. The interpolation weight for the bigram model is included in the weight of the transition $q_A \rightarrow q_B$.

The epsilon transitions into q_U make this WFSA non-deterministic. Consider the score for the sequence (a, b, b). The initial state is q_U, so the symbol a is generated with score $p_1(a)$[3] Next, we can generate b from the unigram model by taking the transition $q_A \rightarrow q_B$, with score $\lambda_2 p_2(b \mid a)$. Alternatively, we can take a transition back to q_U with score λ_1, and then emit b from the unigram model with score $p_1(b)$. To generate the final b token, we face the same choice: emit it directly from the self-transition to q_B, or transition to q_U first.

The total score for the sequence (a, b, b) is the semiring sum over all accepting paths,

$$\begin{aligned} s(a,b,b) = &\big(p_1(a) \otimes \lambda_2 p_2(b \mid a) \otimes \lambda_2 p(b \mid b)\big) \\ &\oplus \big(p_1(a) \otimes \lambda_1 \otimes p_1(b) \otimes \lambda_2 p(b \mid b)\big) \\ &\oplus \big(p_1(a) \otimes \lambda_2 p_2(b \mid a) \otimes p_1(b) \otimes p_1(b)\big) \\ &\oplus \big(p_1(a) \otimes \lambda_1 \otimes p_1(b) \otimes p_1(b) \otimes p_1(b)\big) . \end{aligned} \quad [9.11]$$

Each line in Equation 9.11 represents the probability of a specific path through the WFSA. In the probability semiring, $\otimes$ is multiplication, so that each path is the product of each

[3]We could model the sequence-initial bigram probability $p_2(a \mid \square)$, but for simplicity the WFSA does not admit this possibility, which would require another state.

transition weight, which are themselves probabilities. The $\oplus$ operator is addition, so that the total score is the sum of the scores (probabilities) for each path. This corresponds to the probability under the interpolated bigram language model.

9.1.4 Finite state transducers

Finite state acceptors can determine whether a string is in a regular language, and weighted finite state acceptors can compute a score for every string over a given alphabet. **Finite state transducers** (FSTs) extend the formalism further, by adding an output symbol to each transition. Formally, a finite state transducer is a tuple $T = (Q, \Sigma, \Omega, \lambda, \rho, \delta)$, with Ω representing an output vocabulary and the transition function $\delta : Q \times (\Sigma \cup \epsilon) \times (\Omega \cup \epsilon) \times Q \rightarrow \mathbb{R}$ mapping from states, input symbols, and output symbols to states. The remaining elements $(Q, \Sigma, \lambda, \rho)$ are identical to their definition in weighted finite state acceptors (§ 9.1.3). Thus, each path through the FST T transduces the input string into an output.

String edit distance

The **edit distance** between two strings s and t is a measure of how many operations are required to transform one string into another. There are several ways to compute edit distance, but one of the most popular is the Levenshtein edit distance, which counts the minimum number of insertions, deletions, and substitutions. This can be computed by a one-state weighted finite state transducer, in which the input and output alphabets are identical. For simplicity, consider the alphabet $\Sigma = \Omega = \{a, b\}$. The edit distance can be computed by a one-state transducer with the following transitions,

$$\delta(q, a, a, q) = \delta(q, b, b, q) = 0 \quad [9.12]$$

$$\delta(q, a, b, q) = \delta(q, b, a, q) = 1 \quad [9.13]$$

$$\delta(q, a, \epsilon, q) = \delta(q, b, \epsilon, q) = 1 \quad [9.14]$$

$$\delta(q, \epsilon, a, q) = \delta(q, \epsilon, b, q) = 1. \quad [9.15]$$

The state diagram is shown in Figure 9.5.

For a given string pair, there are multiple paths through the transducer: the best-scoring path from *dessert* to *desert* involves a single deletion, for a total score of 1; the worst-scoring path involves seven deletions and six additions, for a score of 13.

The Porter stemmer

The Porter (1980) stemming algorithm is a "lexicon-free" algorithm for stripping suffixes from English words, using a sequence of character-level rules. Each rule can be described

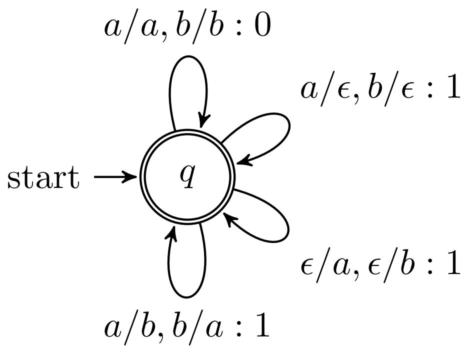


Figure 9.5: State diagram for the Levenshtein edit distance finite state transducer. The label $x/y : c$ indicates a cost of c for a transition with input x and output y.

by an unweighted finite state transducer. The first rule is:

$$\textit{-sses} \rightarrow \textit{-ss} \quad \text{e.g., } \textit{dresses} \rightarrow \textit{dress} \qquad [9.16]$$

$$\textit{-ies} \rightarrow \textit{-i} \quad \text{e.g., } \textit{parties} \rightarrow \textit{parti} \qquad [9.17]$$

$$\textit{-ss} \rightarrow \textit{-ss} \quad \text{e.g., } \textit{dress} \rightarrow \textit{dress} \qquad [9.18]$$

$$\textit{-s} \rightarrow \epsilon \quad \text{e.g., } \textit{cats} \rightarrow \textit{cat} \qquad [9.19]$$

The final two lines appear to conflict; they are meant to be interpreted as an instruction to remove a terminal *-s* unless it is part of an *-ss* ending. A state diagram to handle just these final two lines is shown in Figure 9.6. Make sure you understand how this finite state transducer handles *cats*, *steps*, *bass*, and *basses*.

Inflectional morphology

In **inflectional morphology**, word **lemmas** are modified to add grammatical information such as tense, number, and case. For example, many English nouns are pluralized by the suffix *-s*, and many verbs are converted to past tense by the suffix *-ed*. English's inflectional morphology is considerably simpler than many of the world's languages. For example, Romance languages (derived from Latin) feature complex systems of verb suffixes which must agree with the person and number of the verb, as shown in Table 9.1.

The task of morphological analysis is to read a form like *canto*, and output an analysis like CANTAR+VERB+PRESIND+1P+SING, where +PRESIND describes the tense as present indicative, +1P indicates the first-person, and +SING indicates the singular number. The task of morphological generation is the reverse, going from CANTAR+VERB+PRESIND+1P+SING to *canto*. Finite state transducers are an attractive solution, because they can solve both problems with a single model (Beesley and Karttunen, 2003). As an example, Figure 9.7 shows a fragment of a finite state transducer for Spanish inflectional morphology. The

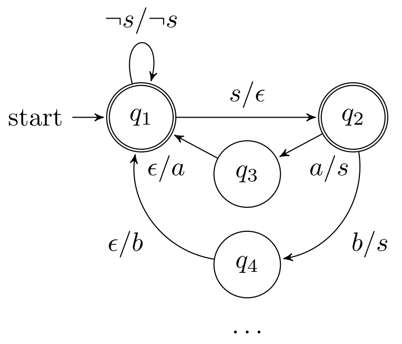


Figure 9.6: State diagram for final two lines of step 1a of the Porter stemming diagram. States q_3 and q_4 "remember" the observations a and b respectively; the ellipsis ... represents additional states for each symbol in the input alphabet. The notation $\neg s/\neg s$ is not part of the FST formalism; it is a shorthand to indicate a set of self-transition arcs for every input/output symbol except s.

infinitive	cantar (to sing)	comer (to eat)	vivir (to live)
yo (1st singular)	canto	como	vivo
tu (2nd singular)	cantas	comes	vives
él, ella, usted (3rd singular)	canta	come	vive
nosotros (1st plural)	cantamos	comemos	vivimos
vosotros (2nd plural, informal)	cantáis	coméis	vivís
ellos, ellas (3rd plural); ustedes (2nd plural)	cantan	comen	viven

Table 9.1: Spanish verb inflections for the present indicative tense. Each row represents a person and number, and each column is a regular example from a class of verbs, as indicated by the ending of the infinitive form.

input vocabulary Σ corresponds to the set of letters used in Spanish spelling, and the output vocabulary Ω corresponds to these same letters, plus the vocabulary of morphological features (e.g., +SING, +VERB). In Figure 9.7, there are two paths that take *canto* as input, corresponding to the verb and noun meanings; the choice between these paths could be guided by a part-of-speech tagger. By **inversion**, the inputs and outputs for each transition are switched, resulting in a finite state generator, capable of producing the correct **surface form** for any morphological analysis.

Finite state morphological analyzers and other unweighted transducers can be designed by hand. The designer's goal is to avoid **overgeneration** — accepting strings or making transductions that are not valid in the language — as well as **undergeneration**

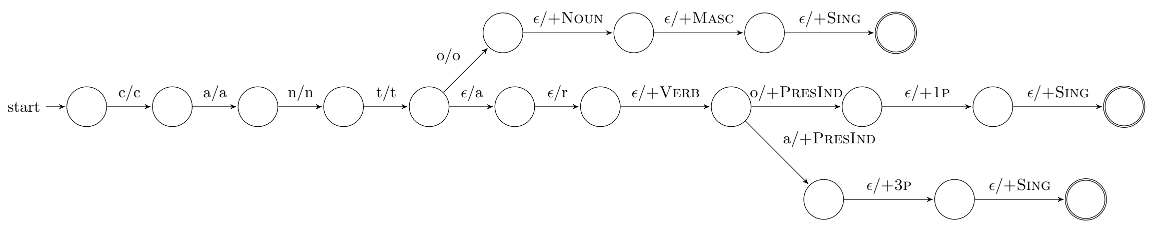


Figure 9.7: Fragment of a finite state transducer for Spanish morphology. There are two accepting paths for the input *canto*: *canto*+NOUN+MASC+SING (masculine singular noun, meaning a song), and *cantar*+VERB+PRESIND+1P+SING (I sing). There is also an accepting path for *canta*, with output *cantar*+VERB+PRESIND+3P+SING (he/she sings).

— failing to accept strings or transductions that are valid. For example, a pluralization transducer that does not accept *foot/feet* would undergenerate. Suppose we "fix" the transducer to accept this example, but as a side effect, it now accepts *boot/beet*; the transducer would then be said to overgenerate. If a transducer accepts *foot/foots* but not *foot/feet*, then it simultaneously overgenerates and undergenerates.

Finite state composition

Designing finite state transducers to capture the full range of morphological phenomena in any real language is a huge task. Modularization is a classic computer science approach for this situation: decompose a large and unwieldly problem into a set of subproblems, each of which will hopefully have a concise solution. Finite state automata can be modularized through **composition**: feeding the output of one transducer T_1 as the input to another transducer T_2, written $T_2 \circ T_1$. Formally, if there exists some y such that $(x, y) \in T_1$ (meaning that T_1 produces output y on input x), and $(y, z) \in T_2$, then $(x, z) \in (T_2 \circ T_1)$. Because finite state transducers are closed under composition, there is guaranteed to be a single finite state transducer that $T_3 = T_2 \circ T_1$, which can be constructed as a machine with one state for each pair of states in T_1 and T_2 (Mohri et al., 2002).

Example: Morphology and orthography In English morphology, the suffix *-ed* is added to signal the past tense for many verbs: *cook→cooked*, *want→wanted*, etc. However, English **orthography** dictates that this process cannot produce a spelling with consecutive *e*'s, so that *bake→baked*, not *bakeed*. A modular solution is to build separate transducers for morphology and orthography. The morphological transducer T_M transduces from *bake*+PAST to *bake+ed*, with the + symbol indicating a segment boundary. The input alphabet of T_M includes the lexicon of words and the set of morphological features; the output alphabet includes the characters *a-z* and the + boundary marker. Next, an orthographic transducer T_O is responsible for the transductions *cook+ed* $\rightarrow$ *cooked*, and *bake+ed* $\rightarrow$ *baked*. The input alphabet of T_O must be the same as the output alphabet for T_M, and the output alphabet is simply the characters *a-z*. The composed transducer $(T_O \circ T_M)$ then transduces from *bake*+PAST to the spelling *baked*. The design of T_O is left as an exercise.

Example: Hidden Markov models Hidden Markov models (chapter 7) can be viewed as weighted finite state transducers, and they can be constructed by transduction. Recall that a hidden Markov model defines a joint probability over words and tags, $\mathrm{p}(\boldsymbol{w}, \boldsymbol{y})$, which can be computed as a path through a **trellis** structure. This trellis is itself a weighted finite state acceptor, with edges between all adjacent nodes $q_{m-1,i} \to q_{m,j}$ on input $Y_m = j$. The edge weights are log-probabilities,

$$\delta(q_{m-1,i}, Y_m = j, q_{m,j}) = \log \mathrm{p}(w_m, Y_m = j \mid Y_{m-i} = j) \quad [9.20]$$

$$= \log \mathrm{p}(w_m \mid Y_m = j) + \log \Pr(Y_m = j \mid Y_{m-1} = i). \quad [9.21]$$

Because there is only one possible transition for each tag Y_m, this WFSA is deterministic. The score for any tag sequence $\{y_m\}_{m=1}^{M}$ is the sum of these log-probabilities, corresponding to the total log probability $\log \mathrm{p}(\boldsymbol{w}, \boldsymbol{y})$. Furthermore, the trellis can be constructed by the composition of simpler FSTs.

- First, construct a "transition" transducer to represent a bigram probability model over tag sequences, T_T. This transducer is almost identical to the n-gram language model acceptor in § 9.1.3: there is one state for each tag, and the edge weights equal to the transition log-probabilities, $\delta(q_i, j, j, q_j) = \log \Pr(Y_m = j \mid Y_{m-1} = i)$. Note that T_T is a transducer, with identical input and output at each arc; this makes it possible to compose T_T with other transducers.
- Next, construct an "emission" transducer to represent the probability of words given tags, T_E. This transducer has only a single state, with arcs for each word/tag pair, $\delta(q_0, i, j, q_0) = \log \Pr(W_m = j \mid Y_m = i)$. The input vocabulary is the set of all tags, and the output vocabulary is the set of all words.
- The composition $T_E \circ T_T$ is a finite state transducer with one state per tag, as shown in Figure 9.8. Each state has $V \times K$ outgoing edges, representing transitions to each of the K other states, with outputs for each of the V words in the vocabulary. The weights for these edges are equal to,

$$\delta(q_i, Y_m = j, w_m, q_j) = \log \mathrm{p}(w_m, Y_m = j \mid Y_{m-1} = i). \quad [9.22]$$

- The trellis is a structure with $M \times K$ nodes, for each of the M words to be tagged and each of the K tags in the tagset. It can be built by composition of $(T_E \circ T_T)$ against an unweighted **chain FSA** $M_A(\boldsymbol{w})$ that is specially constructed to accept only a given input $w_1, w_2, \ldots, w_M$, shown in Figure 9.9. The trellis for input $\boldsymbol{w}$ is built from the composition $M_A(\boldsymbol{w}) \circ (T_E \circ T_T)$. Composing with the unweighted $M_A(\boldsymbol{w})$ does not affect the edge weights from $(T_E \circ T_T)$, but it selects the subset of paths that generate the word sequence $\boldsymbol{w}$.

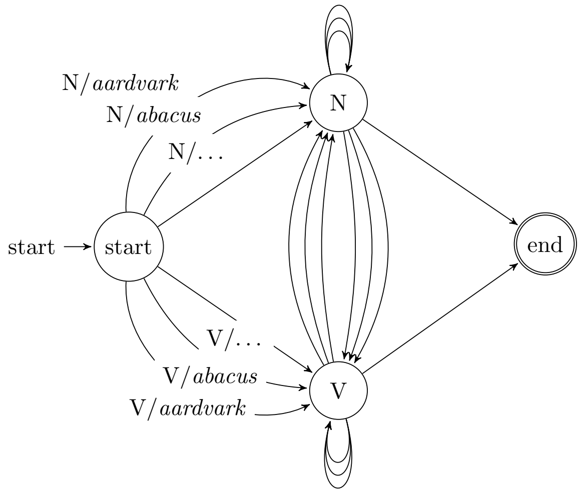


Figure 9.8: Finite state transducer for hidden Markov models, with a small tagset of **n**ouns and **v**erbs. For each pair of tags (including self-loops), there is an edge for every word in the vocabulary. For simplicity, input and output are only shown for the edges from the start state. Weights are also omitted from the diagram; for each edge from q_i to q_j, the weight is equal to $\log \text{p}(w_m, Y_m = j \mid Y_{m-1} = i)$, except for edges to the end state, which are equal to $\log \Pr(Y_m = \blacklozenge \mid Y_{m-1} = i)$.

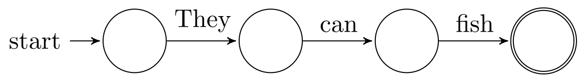


Figure 9.9: Chain finite state acceptor for the input *They can fish*.

9.1.5 *Learning weighted finite state automata

In generative models such as n-gram language models and hidden Markov models, the edge weights correspond to log probabilities, which can be obtained from relative frequency estimation. However, in other cases, we wish to learn the edge weights from input/output pairs. This is difficult in non-deterministic finite state automata, because we do not observe the specific arcs that are traversed in accepting the input, or in transducing from input to output. The path through the automaton is a **latent variable**.

Chapter 5 presented one method for learning with latent variables: expectation maximization (EM). This involves computing a distribution $q(\cdot)$ over the latent variable, and iterating between updates to this distribution and updates to the parameters — in this case, the arc weights. The **forward-backward algorithm** (§ 7.5.3) describes a dynamic program for computing a distribution over arcs in the trellis structure of a hidden Markov model, but this is a special case of the more general problem for finite state automata. Eisner (2002) describes an **expectation semiring**, which enables the expected number of transitions across each arc to be computed through a semiring shortest-path algorithm. Alternative approaches for generative models include Markov Chain Monte Carlo (Chiang et al., 2010) and spectral learning (Balle et al., 2011).

Further afield, we can take a perceptron-style approach, with each arc corresponding to a feature. The classic perceptron update would update the weights by subtracting the difference between the feature vector corresponding to the predicted path and the feature vector corresponding to the correct path. Since the path is not observed, we resort to a **latent variable perceptron**. The model is described formally in § 12.4, but the basic idea is to compute an update from the difference between the features from the predicted path and the features for the best-scoring path that generates the correct output.

9.2 Context-free languages

Beyond the class of regular languages lie the context-free languages. An example of a language that is context-free but not finite state is the set of arithmetic expressions with balanced parentheses. Intuitively, to accept only strings in this language, an FSA would have to "count" the number of left parentheses, and make sure that they are balanced against the number of right parentheses. An arithmetic expression can be arbitrarily long, yet by definition an FSA has a finite number of states. Thus, for any FSA, there will be a string with too many parentheses to count. More formally, the **pumping lemma** is a proof technique for showing that languages are not regular. It is typically demonstrated for the simpler case $a^n b^n$, the language of strings containing a sequence of a's, and then an equal-length sequence of b's.[4]

There are at least two arguments for the relevance of non-regular formal languages to linguistics. First, there are natural language phenomena that are argued to be isomorphic to $a^n b^n$. For English, the classic example is **center embedding**, shown in Figure 9.10. The initial expression *the dog* specifies a single dog. Embedding this expression into *the cat ___ chased* specifies a particular cat — the one chased by the dog. This cat can then be embedded again to specify a goat, in the less felicitous but arguably grammatical expression, *the goat the cat the dog chased kissed*, which refers to the goat who was kissed by the cat which was chased by the dog. Chomsky (1957) argues that to be grammatical, a center-embedded construction must be balanced: if it contains n noun phrases (e.g., *the cat*), they must be followed by exactly $n - 1$ verbs. An FSA that could recognize such expressions would also be capable of recognizing the language $a^n b^n$. Because we can prove that no FSA exists for $a^n b^n$, no FSA can exist for center embedded constructions either. En-

[4]Details of the proof can be found in an introductory computer science theory textbook (e.g., Sipser, 2012).

		the dog		
	the cat	the dog	chased	
the goat	the cat	the dog	chased	kissed
		. . .		

Figure 9.10: Three levels of center embedding

glish includes center embedding, and so the argument goes, English grammar as a whole cannot be regular.[5]

A more practical argument for moving beyond regular languages is modularity. Many linguistic phenomena — especially in syntax — involve constraints that apply at long distance. Consider the problem of determiner-noun number agreement in English: we can say *the coffee* and *these coffees*, but not **these coffee*. By itself, this is easy enough to model in an FSA. However, fairly complex modifying expressions can be inserted between the determiner and the noun:

(9.2)
a. the burnt coffee
b. the badly-ground coffee
c. the burnt and badly-ground Italian coffee
d. these burnt and badly-ground Italian coffees
e. * these burnt and badly-ground Italian coffee

Again, an FSA can be designed to accept modifying expressions such as *burnt and badly-ground Italian*. Let's call this FSA F_M. To reject the final example, a finite state acceptor must somehow "remember" that the determiner was plural when it reaches the noun *coffee* at the end of the expression. The only way to do this is to make two identical copies of F_M: one for singular determiners, and one for plurals. While this is possible in the finite state framework, it is inconvenient — especially in languages where more than one attribute of the noun is marked by the determiner. **Context-free languages** facilitate modularity across such long-range dependencies.

9.2.1 Context-free grammars

Context-free languages are specified by **context-free grammars** (CFGs), which are tuples (N, Σ, R, S) consisting of:

[5]The claim that arbitrarily deep center-embedded expressions are grammatical has drawn skepticism. Corpus evidence shows that embeddings of depth greater than two are exceedingly rare (Karlsson, 2007), and that embeddings of depth greater than three are completely unattested. If center-embedding is capped at some finite depth, then it is regular.

$$\begin{aligned}\text{S} &\rightarrow \text{S Op S} \mid \text{Num}\\ \text{Op} &\rightarrow + \mid - \mid \times \mid \div\\ \text{Num} &\rightarrow \text{Num Digit} \mid \text{Digit}\\ \text{Digit} &\rightarrow \texttt{0} \mid \texttt{1} \mid \texttt{2} \mid \ldots \mid \texttt{9}\end{aligned}$$

Figure 9.11: A context-free grammar for arithmetic expressions

- a finite set of **non-terminals** N;
- a finite alphabet Σ of **terminal symbols**;
- a set of **production rules** R, each of the form $A \rightarrow \beta$, where $A \in N$ and $\beta \in (\Sigma \cup N)^*$;
- a designated start symbol S.

In the production rule $A \rightarrow \beta$, the left-hand side (LHS) A must be a non-terminal; the right-hand side (RHS) can be a sequence of terminals or non-terminals, $\{n, \sigma\}^*, n \in N, \sigma \in \Sigma$. A non-terminal can appear on the left-hand side of many production rules. A non-terminal can appear on both the left-hand side and the right-hand side; this is a **recursive production**, and is analogous to self-loops in finite state automata. The name "context-free" is based on the property that the production rule depends only on the LHS, and not on its ancestors or neighbors; this is analogous to Markov property of finite state automata, in which the behavior at each step depends only on the current state, and not on the path by which that state was reached.

A **derivation** τ is a sequence of steps from the start symbol S to a surface string $\boldsymbol{w} \in \Sigma^*$, which is the **yield** of the derivation. A string $\boldsymbol{w}$ is in a context-free language if there is some derivation from S yielding $\boldsymbol{w}$. **Parsing** is the problem of finding a derivation for a string in a grammar. Algorithms for parsing are described in chapter 10.

Like regular expressions, context-free grammars define the language but not the computation necessary to recognize it. The context-free analogues to finite state acceptors are **pushdown automata**, a theoretical model of computation in which input symbols can be pushed onto a stack with potentially infinite depth. For more details, see Sipser (2012).

Example

Figure 9.11 shows a context-free grammar for arithmetic expressions such as $1 + 2 \div 3 - 4$. In this grammar, the terminal symbols include the digits {`1, 2, ..., 9`} and the operators $\{+, -, \times, \div\}$. The rules include the | symbol, a notational convenience that makes it possible to specify multiple right-hand sides on a single line: the statement $A \rightarrow x \mid y$

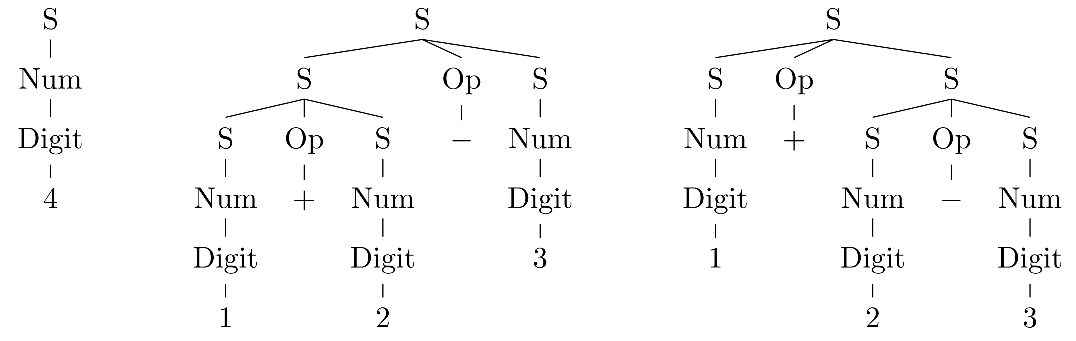


Figure 9.12: Some example derivations from the arithmetic grammar in Figure 9.11

defines *two* productions, $A \to x$ and $A \to y$. This grammar is recursive: the non-termals S and NUM can produce themselves.

Derivations are typically shown as trees, with production rules applied from the top to the bottom. The tree on the left in Figure 9.12 describes the derivation of a single digit, through the sequence of productions S $\to$ NUM $\to$ DIGIT $\to$ 4 (these are all **unary productions**, because the right-hand side contains a single element). The other two trees in Figure 9.12 show alternative derivations of the string 1 + 2 − 3. The existence of multiple derivations for a string indicates that the grammar is **ambiguous**.

Context-free derivations can also be written out according to the pre-order tree traversal.[6] For the two derivations of `1 + 2 - 3` in Figure 9.12, the notation is:

(S (S (S (Num (Digit `1`))) (Op +) (S (Num (Digit `2`)))) (Op −) (S (Num (Digit `3`)))) [9.23]

(S (S (Num (Digit `1`))) (Op +) (S (Num (Digit `2`)) (Op −) (S (Num (Digit `3`))))). [9.24]

Grammar equivalence and Chomsky Normal Form

A single context-free language can be expressed by more than one context-free grammar. For example, the following two grammars both define the language $a^n b^n$ for $n > 0$.

$$S \to aSb \mid ab$$
$$S \to aSb \mid aabb \mid ab$$

Two grammars are **weakly equivalent** if they generate the same strings. Two grammars are **strongly equivalent** if they generate the same strings via the same derivations. The grammars above are only weakly equivalent.

[6]This is a depth-first left-to-right search that prints each node the first time it is encountered (Cormen et al., 2009, chapter 12).

In **Chomsky Normal Form (CNF)**, the right-hand side of every production includes either two non-terminals, or a single terminal symbol:

$$\begin{aligned} A \rightarrow & BC \\ A \rightarrow & a \end{aligned}$$

All CFGs can be converted into a CNF grammar that is weakly equivalent. To convert a grammar into CNF, we first address productions that have more than two non-terminals on the RHS by creating new "dummy" non-terminals. For example, if we have the production,

$$\text{W} \rightarrow \text{X Y Z}, \tag{9.25}$$

it is replaced with two productions,

$$\text{W} \rightarrow \text{X W}\backslash\text{X} \tag{9.26}$$

$$\text{W}\backslash\text{X} \rightarrow \text{Y Z}. \tag{9.27}$$

In these productions, W\X is a new dummy non-terminal. This transformation **binarizes** the grammar, which is critical for efficient bottom-up parsing, as we will see in chapter 10. Productions whose right-hand side contains a mix of terminal and non-terminal symbols can be replaced in a similar fashion.

Unary non-terminal productions $A \rightarrow B$ are replaced as follows: for each production $B \rightarrow \alpha$ in the grammar, add a new production $A \rightarrow \alpha$. For example, in the grammar described in Figure 9.11, we would replace NUM → DIGIT with NUM → `1` | `2` | ... | `9`. However, we keep the production NUM → NUM DIGIT, which is a valid binary production.

9.2.2 Natural language syntax as a context-free language

Context-free grammars can be used to represent **syntax**, which is the set of rules that determine whether an utterance is judged to be grammatical. If this representation were perfectly faithful, then a natural language such as English could be transformed into a formal language, consisting of exactly the (infinite) set of strings that would be judged to be grammatical by a fluent English speaker. We could then build parsing software that would automatically determine if a given utterance were grammatical.[7]

Contemporary theories generally do *not* consider natural languages to be context-free (see § 9.3), yet context-free grammars are widely used in natural language parsing. The reason is that context-free representations strike a good balance: they cover a broad range of syntactic phenomena, and they can be parsed efficiently. This section therefore describes how to handle a core fragment of English syntax in context-free form, following

[7]To move beyond this cursory treatment of syntax, consult the short introductory manuscript by Bender (2013), or the longer text by Akmajian et al. (2010).

the conventions of the **Penn Treebank** (PTB; Marcus et al., 1993), a large-scale annotation of English language syntax. The generalization to "mildly" context-sensitive languages is discussed in § 9.3.

The Penn Treebank annotation is a **phrase-structure grammar** of English. This means that sentences are broken down into **constituents**, which are contiguous sequences of words that function as coherent units for the purpose of linguistic analysis. Constituents generally have a few key properties:

Movement. Constituents can often be moved around sentences as units.

(9.3) a. Abigail gave (her brother) (a fish).
b. Abigail gave (a fish) to (her brother).

In contrast, *gave her* and *brother a* cannot easily be moved while preserving grammaticality.

Substitution. Constituents can be substituted by other phrases of the same type.

(9.4) a. Max thanked (his older sister).
b. Max thanked (her).

In contrast, substitution is not possible for other contiguous units like *Max thanked* and *thanked his*.

Coordination. Coordinators like *and* and *or* can conjoin constituents.

(9.5) a. (Abigail) and (her younger brother) bought a fish.
b. Abigail (bought a fish) and (gave it to Max).
c. Abigail (bought) and (greedily ate) a fish.

Units like *brother bought* and *bought a* cannot easily be coordinated.

These examples argue for units such as *her brother* and *bought a fish* to be treated as constituents. Other sequences of words in these examples, such as *Abigail gave* and *brother a fish*, cannot be moved, substituted, and coordinated in these ways. In phrase-structure grammar, constituents are nested, so that *the senator from New Jersey* contains the constituent *from New Jersey*, which in turn contains *New Jersey*. The sentence itself is the maximal constituent; each word is a minimal constituent, derived from a unary production from a part-of-speech tag. Between part-of-speech tags and sentences are **phrases**. In phrase-structure grammar, phrases have a type that is usually determined by their **head word**: for example, a **noun phrase** corresponds to a noun and the group of words that modify it, such as *her younger* ***brother***; a **verb phrase** includes the verb and its modifiers, such as ***bought*** *a fish* and *greedily* ***ate*** *it*.

In context-free grammars, each phrase type is a non-terminal, and each constituent is the substring that the non-terminal yields. Grammar design involves choosing the right set of non-terminals. Fine-grained non-terminals make it possible to represent more fine-grained linguistic phenomena. For example, by distinguishing singular and plural noun phrases, it is possible to have a grammar of English that generates only sentences that obey subject-verb agreement. However, enforcing subject-verb agreement is considerably more complicated in languages like Spanish, where the verb must agree in both person and number with subject. In general, grammar designers must trade off between **over-generation** — a grammar that permits ungrammatical sentences — and **undergeneration** — a grammar that fails to generate grammatical sentences. Furthermore, if the grammar is to support manual annotation of syntactic structure, it must be simple enough to annotate efficiently.

9.2.3 A phrase-structure grammar for English

To better understand how phrase-structure grammar works, let's consider the specific case of the Penn Treebank grammar of English. The main phrase categories in the Penn Treebank (PTB) are based on the main part-of-speech classes: noun phrase (NP), verb phrase (VP), prepositional phrase (PP), adjectival phrase (ADJP), and adverbial phrase (ADVP). The top-level category is S, which conveniently stands in for both "sentence" and the "start" symbol. **Complement clauses** (e.g., *I take the good old fashioned ground* ***that the whale is a fish***) are represented by the non-terminal SBAR. The terminal symbols in the grammar are individual words, which are generated from unary productions from part-of-speech tags (the PTB tagset is described in § 8.1).

This section describes some of the most common productions from the major phrase-level categories, explaining how to generate individual tag sequences. The production rules are approached in a "theory-driven" manner: first the syntactic properties of each phrase type are described, and then some of the necessary production rules are listed. But it is important to keep in mind that the Penn Treebank was produced in a "data-driven" manner. After the set of non-terminals was specified, annotators were free to analyze each sentence in whatever way seemed most linguistically accurate, subject to some high-level guidelines. The grammar of the Penn Treebank is simply the set of productions that were required to analyze the several million words of the corpus. By design, the grammar overgenerates — it does not exclude ungrammatical sentences. Furthermore, while the productions shown here cover some of the most common cases, they are only a small fraction of the several thousand different types of productions in the Penn Treebank.

Sentences

The most common production rule for sentences is,

$$\text{S} \rightarrow \text{NP VP} \tag{9.28}$$

which accounts for simple sentences like *Abigail ate the kimchi* — as we will see, the direct object *the kimchi* is part of the verb phrase. But there are more complex forms of sentences as well:

S →ADVP NP VP	*Unfortunately Abigail ate the kimchi.*	[9.29]
S →S CC S	*Abigail ate the kimchi and Max had a burger.*	[9.30]
S →VP	*Eat the kimchi.*	[9.31]

where ADVP is an adverbial phrase (e.g., *unfortunately*, *very unfortunately*) and CC is a coordinating conjunction (e.g., *and*, *but*).[8]

Noun phrases

Noun phrases refer to entities, real or imaginary, physical or abstract: *Asha*, *the steamed dumpling*, *parts and labor*, *nobody*, *the whiteness of the whale*, and *the rise of revolutionary syndicalism in the early twentieth century.* Noun phrase productions include "bare" nouns, which may optionally follow determiners, as well as pronouns:

$$\text{NP} \rightarrow \text{NN} \mid \text{NNS} \mid \text{NNP} \mid \text{PRP} \tag{9.32}$$
$$\text{NP} \rightarrow \text{DET NN} \mid \text{DET NNS} \mid \text{DET NNP} \tag{9.33}$$

The tags NN, NNS, and NNP refer to singular, plural, and proper nouns; PRP refers to personal pronouns, and DET refers to determiners. The grammar also contains terminal productions from each of these tags, e.g., PRP → *I* | *you* | *we* | . . . .

Noun phrases may be modified by adjectival phrases (ADJP; e.g., *the small Russian dog*) and numbers (CD; e.g., *the five pastries*), each of which may optionally follow a determiner:

$$\text{NP} \rightarrow \text{ADJP NN} \mid \text{ADJP NNS} \mid \text{DET ADJP NN} \mid \text{DET ADJP NNS} \tag{9.34}$$
$$\text{NP} \rightarrow \text{CD NNS} \mid \text{DET CD NNS} \mid \ldots \tag{9.35}$$

Some noun phrases include multiple nouns, such as *the liberation movement* and *an antelope horn*, necessitating additional productions:

$$\text{NP} \rightarrow \text{NN NN} \mid \text{NN NNS} \mid \text{DET NN NN} \mid \ldots \tag{9.36}$$

[8]Notice that the grammar does not include the recursive production S → ADVP S. It may be helpful to think about why this production would cause the grammar to overgenerate.

These multiple noun constructions can be combined with adjectival phrases and cardinal numbers, leading to a large number of additional productions.

Recursive noun phrase productions include coordination, prepositional phrase attachment, subordinate clauses, and verb phrase adjuncts:

NP →NP CC NP	*e.g., the red and the black*	[9.37]
NP →NP PP	*e.g., the President of the Georgia Institute of Technology*	[9.38]
NP →NP SBAR	*e.g., a whale which he had wounded*	[9.39]
NP →NP VP	*e.g., a whale taken near Shetland*	[9.40]

These recursive productions are a major source of ambiguity, because the VP and PP non-terminals can also generate NP children. Thus, the *the President of the Georgia Institute of Technology* can be derived in two ways, as can *a whale taken near Shetland in October*.

But aside from these few recursive productions, the noun phrase fragment of the Penn Treebank grammar is relatively flat, containing a large of number of productions that go from NP directly to a sequence of parts-of-speech. If noun phrases had more internal structure, the grammar would need fewer rules, which, as we will see, would make parsing faster and machine learning easier. Vadas and Curran (2011) propose to add additional structure in the form of a new non-terminal called a **nominal modifier** (NML), e.g.,

(9.6) a. (NP (NN crude) (NN oil) (NNS prices)) (PTB analysis)
b. (NP (NML (NN crude) (NN oil)) (NNS prices)) (NML-style analysis).

Another proposal is to treat the determiner as the head of a **determiner phrase** (DP; Abney, 1987). There are linguistic arguments for and against determiner phrases (e.g., Van Eynde, 2006). From the perspective of context-free grammar, DPs enable more structured analyses of some constituents, e.g.,

(9.7) a. (NP (DT the) (JJ white) (NN whale)) (PTB analysis)
b. (DP (DT the) (NP (JJ white) (NN whale))) (DP-style analysis).

Verb phrases

Verb phrases describe actions, events, and states of being. The PTB tagset distinguishes several classes of verb inflections: base form (VB; *she likes to snack*), present-tense third-person singular (VBZ; *she snacks*), present tense but not third-person singular (VBP; *they snack*), past tense (VBD; *they snacked*), present participle (VBG; *they are snacking*), and past participle (VBN; *they had snacked*).[9] Each of these forms can constitute a verb phrase on its

[9]This tagset is specific to English: for example, VBP is a meaningful category only because English morphology distinguishes third-person singular from all person-number combinations.

own:

VP → VB | VBZ | VBD | VBN | VBG | VBP [9.41]

More complex verb phrases can be formed by a number of recursive productions, including the use of coordination, modal verbs (MD; *she should snack*), and the infinitival *to* (TO):

VP → MD VP	*She **will snack***	[9.42]
VP → VBD VP	*She **had snacked***	[9.43]
VP → VBZ VP	*She **has been snacking***	[9.44]
VP → VBN VP	*She has **been snacking***	[9.45]
VP → TO VP	*She wants **to snack***	[9.46]
VP → VP CC VP	*She **buys and eats** many snacks*	[9.47]

Each of these productions uses recursion, with the VP non-terminal appearing in both the LHS and RHS. This enables the creation of complex verb phrases, such as *She will have wanted to have been snacking*.

Transitive verbs take noun phrases as direct objects, and ditransitive verbs take two direct objects:

VP → VBZ NP	*She **teaches algebra***	[9.48]
VP → VBG NP	*She has been **teaching algebra***	[9.49]
VP → VBD NP NP	*She **taught** her brother algebra*	[9.50]

These productions are *not* recursive, so a unique production is required for each verb part-of-speech. They also do not distinguish transitive from intransitive verbs, so the resulting grammar overgenerates examples like **She sleeps sushi* and **She learns Boyang algebra*. Sentences can also be direct objects:

VP → VBZ S	*Hunter **wants to eat the kimchi***	[9.51]
VP → VBZ SBAR	*Hunter **knows that Tristan ate the kimchi***	[9.52]

The first production overgenerates, licensing sentences like **Hunter sees Tristan eats the kimchi*. This problem could be addressed by designing a more specific set of sentence non-terminals, indicating whether the main verb can be conjugated.

Verbs can also be modified by prepositional phrases and adverbial phrases:

VP → VBZ PP	*She **studies at night***	[9.53]
VP → VBZ ADVP	*She **studies intensively***	[9.54]
VP → ADVP VBG	*She is **not studying***	[9.55]

Again, because these productions are not recursive, the grammar must include productions for every verb part-of-speech.

A special set of verbs, known as **copula**, can take **predicative adjectives** as direct objects:

VP → VBZ ADJP	*She **is hungry***	[9.56]
VP → VBP ADJP	*Success **seems increasingly unlikely***	[9.57]

The PTB does not have a special non-terminal for copular verbs, so this production generates non-grammatical examples such as **She eats tall*.

Particles (PRT as a phrase; RP as a part-of-speech) work to create phrasal verbs:

VP → VB PRT	*She told them to **fuck off***	[9.58]
VP → VBD PRT NP	*They **gave up their ill-gotten gains***	[9.59]

As the second production shows, particle productions are required for all configurations of verb parts-of-speech and direct objects.

Other contituents

The remaining constituents require far fewer productions. **Prepositional phrases** almost always consist of a preposition and a noun phrase,

PP → IN NP	*the whiteness **of the whale***	[9.60]
PP → TO NP	*What the white whale was **to Ahab**, has been hinted*	[9.61]

Similarly, complement clauses consist of a complementizer (usually a preposition, possibly null) and a sentence,

SBAR → IN S	*She said **that it was spicy***	[9.62]
SBAR → S	*She said **it was spicy***	[9.63]

Adverbial phrases are usually bare adverbs (ADVP → RB), with a few exceptions:

ADVP → RB RBR	*They went **considerably further***	[9.64]
ADVP → ADVP PP	*They went **considerably further than before***	[9.65]

The tag RBR is a comparative adverb.

Adjectival phrases extend beyond bare adjectives (ADJP → JJ) in a number of ways:

ADJP → RB JJ	*very hungry*	[9.66]
ADJP → RBR JJ	*more hungry*	[9.67]
ADJP → JJS JJ	*best possible*	[9.68]
ADJP → RB JJR	*even bigger*	[9.69]
ADJP → JJ CC JJ	*high and mighty*	[9.70]
ADJP → JJ JJ	*West German*	[9.71]
ADJP → RB VBN	*previously reported*	[9.72]

The tags JJR and JJS refer to comparative and superlative adjectives respectively.

All of these phrase types can be coordinated:

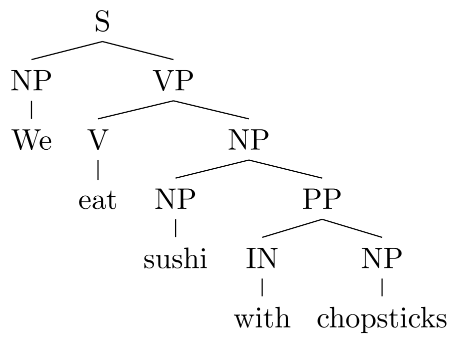
PP →PP CC PP	*on time and under budget*	[9.73]
ADVP →ADVP CC ADVP	*now and two years ago*	[9.74]
ADJP →ADJP CC ADJP	*quaint and rather deceptive*	[9.75]
SBAR →SBAR CC SBAR	*whether they want control or whether they want exports*	[9.76]

9.2.4 Grammatical ambiguity

Context-free parsing is useful not only because it determines whether a sentence is grammatical, but mainly because the constituents and their relations can be applied to tasks such as information extraction (chapter 17) and sentence compression (Jing, 2000; Clarke and Lapata, 2008). However, the **ambiguity** of wide-coverage natural language grammars poses a serious problem for such potential applications. As an example, Figure 9.13 shows two possible analyses for the simple sentence *We eat sushi with chopsticks*, depending on whether the *chopsticks* modify *eat* or *sushi*. Realistic grammars can license thousands or even millions of parses for individual sentences. **Weighted context-free grammars** solve this problem by attaching weights to each production, and selecting the derivation with the highest score. This is the focus of chapter 10.

9.3 *Mildly context-sensitive languages

Beyond context-free languages lie **context-sensitive languages**, in which the expansion of a non-terminal depends on its neighbors. In the general class of context-sensitive languages, computation becomes much more challenging: the membership problem for context-sensitive languages is PSPACE-complete. Since PSPACE contains the complexity class NP (problems that can be solved in polynomial time on a non-deterministic Turing



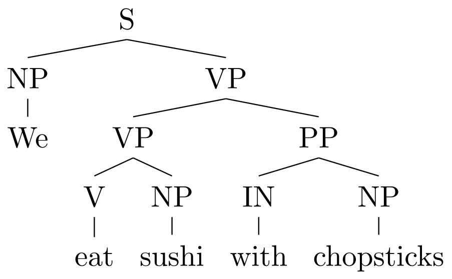


Figure 9.13: Two derivations of the same sentence

machine), PSPACE-complete problems cannot be solved efficiently if P $\neq$ NP. Thus, designing an efficient parsing algorithm for the full class of context-sensitive languages is probably hopeless.[10]

However, Joshi (1985) identifies a set of properties that define **mildly context-sensitive languages**, which are a strict subset of context-sensitive languages. Like context-free languages, mildly context-sensitive languages are parseable in polynomial time. However, the mildly context-sensitive languages include non-context-free languages, such as the "copy language" $\{\boldsymbol{ww} \mid \boldsymbol{w} \in \Sigma^*\}$ and the language $a^m b^n c^m d^n$. Both are characterized by **cross-serial dependencies**, linking symbols at long distance across the string.[11] For example, in the language $a^n b^m c^n d^m$, each a symbol is linked to exactly one c symbol, regardless of the number of intervening b symbols.

9.3.1 Context-sensitive phenomena in natural language

Such phenomena are occasionally relevant to natural language. A classic example is found in Swiss-German (Shieber, 1985), in which sentences such as *we let the children help Hans paint the house* are realized by listing all nouns before all verbs, i.e., *we the children Hans the house let help paint*. Furthermore, each noun's determiner is dictated by the noun's **case marking** (the role it plays with respect to the verb). Using an argument that is analogous to the earlier discussion of center-embedding (§ 9.2), Shieber describes these case marking constraints as a set of cross-serial dependencies, homomorphic to $a^m b^n c^m d^n$, and therefore not context-free.

[10]If PSPACE $\neq$ NP, then it contains problems that cannot be solved in polynomial time on a non-deterministic Turing machine; equivalently, solutions to these problems cannot even be checked in polynomial time (Arora and Barak, 2009).

[11]A further condition of the set of mildly-context-sensitive languages is *constant growth*: if the strings in the language are arranged by length, the gap in length between any pair of adjacent strings is bounded by some language specific constant. This condition excludes languages such as $\{a^{2^n} \mid n \geq 0\}$.

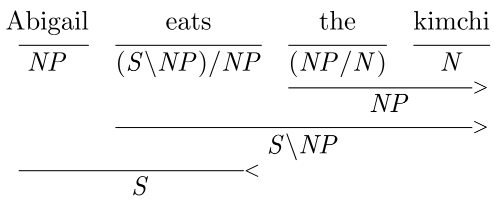


Figure 9.14: A syntactic analysis in CCG involving forward and backward function application

As with the move from regular to context-free languages, mildly context-sensitive languages can also be motivated by expedience. While finite sequences of cross-serial dependencies can in principle be handled in a context-free grammar, it is often more convenient to use a mildly context-sensitive formalism like **tree-adjoining grammar** (TAG) and **combinatory categorial grammar** (CCG). TAG-inspired parsers have been shown to be particularly effective in parsing the Penn Treebank (Collins, 1997; Carreras et al., 2008), and CCG plays a leading role in current research on semantic parsing (Zettlemoyer and Collins, 2005). These two formalisms are weakly equivalent: any language that can be specified in TAG can also be specified in CCG, and vice versa (Joshi et al., 1991). The remainder of the chapter gives a brief overview of CCG, but you are encouraged to consult Joshi and Schabes (1997) and Steedman and Baldridge (2011) for more detail on TAG and CCG respectively.

9.3.2 Combinatory categorial grammar

In combinatory categorial grammar, structural analyses are built up through a small set of generic combinatorial operations, which apply to immediately adjacent sub-structures. These operations act on the categories of the sub-structures, producing a new structure with a new category. The basic categories include S (sentence), NP (noun phrase), VP (verb phrase) and N (noun). The goal is to label the entire span of text as a sentence, S.

Complex categories, or types, are constructed from the basic categories, parentheses, and forward and backward slashes: for example, S/NP is a complex type, indicating a sentence that is lacking a noun phrase to its right; S\NP is a sentence lacking a noun phrase to its left. Complex types act as functions, and the most basic combinatory operations are function application to either the right or left neighbor. For example, the type of a verb phrase, such as *eats*, would be S\NP. Applying this function to a subject noun phrase to its left results in an analysis of *Abigail eats* as category S, indicating a successful parse.

Transitive verbs must first be applied to the direct object, which in English appears to the right of the verb, before the subject, which appears on the left. They therefore have the more complex type (S\NP)/NP. Similarly, the application of a determiner to the noun at

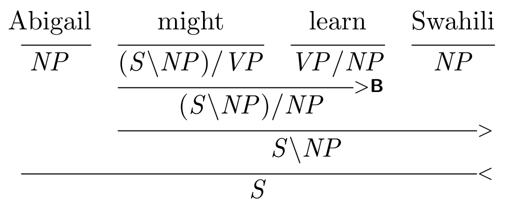


Figure 9.15: A syntactic analysis in CCG involving function composition (example modified from Steedman and Baldridge, 2011)

its right results in a noun phrase, so determiners have the type NP/N. Figure 9.14 provides an example involving a transitive verb and a determiner. A key point from this example is that it can be trivially transformed into phrase-structure tree, by treating each function application as a constituent phrase. Indeed, when CCG's only combinatory operators are forward and backward function application, it is equivalent to context-free grammar. However, the location of the "effort" has changed. Rather than designing good productions, the grammar designer must focus on the **lexicon** — choosing the right categories for each word. This makes it possible to parse a wide range of sentences using only a few generic combinatory operators.

Things become more interesting with the introduction of two additional operators: **composition** and **type-raising**. Function composition enables the combination of complex types: $X/Y \circ Y/Z \Rightarrow_{\mathbf{B}} X/Z$ (forward composition) and $Y\backslash Z \circ X\backslash Y \Rightarrow_{\mathbf{B}} X\backslash Z$ (backward composition).[12] Composition makes it possible to "look inside" complex types, and combine two adjacent units if the "input" for one is the "output" for the other. Figure 9.15 shows how function composition can be used to handle modal verbs. While this sentence can be parsed using only function application, the composition-based analysis is preferable because the unit *might learn* functions just like a transitive verb, as in the example *Abigail studies Swahili*. This in turn makes it possible to analyze conjunctions such as *Abigail studies and might learn Swahili*, attaching the direct object *Swahili* to the entire conjoined verb phrase *studies and might learn*. The Penn Treebank grammar fragment from § 9.2.3 would be unable to handle this case correctly: the direct object *Swahili* could attach only to the second verb *learn*.

Type raising converts an element of type X to a more complex type: $X \Rightarrow_T T/(T\backslash X)$ (forward type-raising to type T), and $X \Rightarrow_T T\backslash(T/X)$ (backward type-raising to type T). Type-raising makes it possible to reverse the relationship between a function and its argument — by transforming the argument into a function over functions over arguments! An example may help. Figure 9.15 shows how to analyze an object relative clause, *a story that Abigail tells*. The problem is that *tells* is a transitive verb, expecting a direct object to its right. As a result, *Abigail tells* is not a valid constituent. The issue is resolved by raising

[12]The subscript $\mathbf{B}$ follows notation from Curry and Feys (1958).

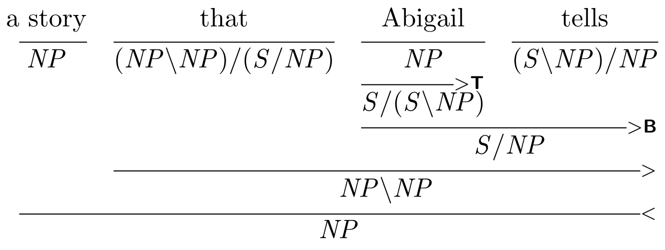


Figure 9.16: A syntactic analysis in CCG involving an object relative clause

Abigail from NP to the complex type (S/NP)\NP). This function can then be combined with the transitive verb *tells* by forward composition, resulting in the type (S/NP), which is a sentence lacking a direct object to its right.[13] From here, we need only design the lexical entry for the complementizer *that* to expect a right neighbor of type (S/NP), and the remainder of the derivation can proceed by function application.

Composition and type-raising give CCG considerable power and flexibility, but at a price. The simple sentence *Abigail tells Max* can be parsed in two different ways: by function application (first forming the verb phrase *tells Max*), and by type-raising and composition (first forming the non-constituent *Abigail tells*). This **derivational ambiguity** does not affect the resulting linguistic analysis, so it is sometimes known as **spurious ambiguity**. Hockenmaier and Steedman (2007) present a translation algorithm for converting the Penn Treebank into CCG derivations, using composition and type-raising only when necessary.

Exercises

1. Sketch out the state diagram for finite-state acceptors for the following languages on the alphabet $\{a, b\}$.

   a) Even-length strings. (Be sure to include 0 as an even number.)

   b) Strings that contain aaa as a substring.

   c) Strings containing an even number of a and an odd number of b symbols.

   d) Strings in which the substring bbb must be terminal if it appears — the string need not contain bbb, but if it does, nothing can come after it.

2. Levenshtein edit distance is the number of insertions, substitutions, or deletions required to convert one string to another.

[13]The missing direct object would be analyzed as a **trace** in CFG-like approaches to syntax, including the Penn Treebank.

a) Define a finite-state acceptor that accepts all strings with edit distance 1 from the target string, *target*.

b) Now think about how to generalize your design to accept all strings with edit distance from the target string equal to d. If the target string has length ℓ, what is the minimal number of states required?

3. Construct an FSA in the style of Figure 9.3, which handles the following examples:

   - *nation*/N, *national*/ADJ, *nationalize*/V, *nationalizer*/N
   - *America*/N, *American*/ADJ, *Americanize*/V, *Americanizer*/N

   Be sure that your FSA does not accept any further derivations, such as **nationalizeral* and **Americanizern*.

4. Show how to construct a trigram language model in a weighted finite-state acceptor. Make sure that you handle the edge cases at the beginning and end of the input.

5. Extend the FST in Figure 9.6 to handle the other two parts of rule 1a of the Porter stemmer: *-sses* $\rightarrow$ *ss*, and *-ies* $\rightarrow$ *-i*.

6. § 9.1.4 describes T_O, a transducer that captures English orthography by transducing *cook* + *ed* $\rightarrow$ *cooked* and *bake* + *ed* $\rightarrow$ *baked*. Design an unweighted finite-state transducer that captures this property of English orthography.

   Next, augment the transducer to appropriately model the suffix *-s* when applied to words ending in *s*, e.g. *kiss+s* $\rightarrow$ *kisses*.

7. Add parenthesization to the grammar in Figure 9.11 so that it is no longer ambiguous.

8. Construct three examples — a noun phrase, a verb phrase, and a sentence — which can be derived from the Penn Treebank grammar fragment in § 9.2.3, yet are not grammatical. Avoid reusing examples from the text. Optionally, propose corrections to the grammar to avoid generating these cases.

9. Produce parses for the following sentences, using the Penn Treebank grammar fragment from § 9.2.3.

   (9.8) This aggression will not stand.

   (9.9) I can get you a toe.

   (9.10) Sometimes you eat the bar and sometimes the bar eats you.

   Then produce parses for three short sentences from a news article from this week.

10. * One advantage of CCG is its flexibility in handling coordination:

    (9.11) a. *Hunter and Tristan speak Hawaiian*
    b. *Hunter speaks and Tristan understands Hawaiian*

    Define the lexical entry for *and* as

    $$\textit{and} := (X/X)\backslash X, \qquad [9.77]$$

    where X can refer to any type. Using this lexical entry, show how to parse the two examples above. In the second example, *Swahili* should be combined with the coordination *Abigail speaks and Max understands*, and not just with the verb *understands*.