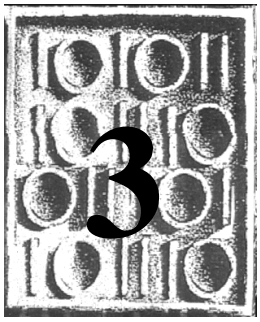
CHAPTER



Grammars and Parsing

3.1 Grammars and Sentence Structure

3.2 What Makes a Good Grammar

3.3 A Top-Down Parser

3.4 A Bottom-Up Chart Parser

3.5 Transition Network Grammars

- 3.6 Top-Down Chart Parsing
- 3.7 Finite State Models and Morphological Processing
- 3.8 Grammars and Logic Programming

To examine how the syntactic structure of a sentence can be computed, you must consider two things: the **grammar**, which is a formal specification of the structures allowable in the language, and the **parsing technique**, which is the method of analyzing a sentence to determine its structure according to the grammar. This chapter examines different ways to specify simple grammars and considers some fundamental parsing techniques. Chapter 4 then describes the methods for constructing syntactic representations that are useful for later semantic interpretation.

The discussion begins by introducing a notation for describing the structure of natural language and describing some naive parsing techniques for that grammar. The second section describes some characteristics of a good grammar. The third section then considers a simple parsing technique and introduces the idea of parsing as a search process. The fourth section describes a method for building efficient parsers using a structure called a chart. The fifth section then describes an alternative representation of grammars based on transition networks. The remaining sections deal with optional and advanced issues. Section 3.6 describes a top-down chart parser that combines the advantages of top-down and bottom-up approaches. Section 3.7 introduces the notion of finite state transducers and discusses their use in morphological processing. Section 3.8 shows how to encode context-free grammars as assertions in PROLOG, introducing the notion of logic grammars.

3.1 Grammars and Sentence Structure

This section considers methods of describing the structure of sentences and explores ways of characterizing all the legal structures in a language. The most common way of representing how a sentence is broken into its major subparts, and how those subparts are broken up in turn, is as a **tree**. The tree representation for the sentence *John ate the cat* is shown in Figure 3.1. This illustration can be read as follows: The sentence (S) consists of an initial noun phrase (NP) and a verb phrase (VP). The initial noun phrase is made of the simple NAME *John*. The verb phrase is composed of a verb (V) *ate* and an NP, which consists of an article (ART) *the* and a common noun (N) *cat*. In list notation this same structure could be represented as

```
(S  (NP  (NAME John))
    (VP  (V ate)
         (NP  (ART the)
              (N cat))))
```

Since trees play such an important role throughout this book, some terminology needs to be introduced. Trees are a special form of graph, which are structures consisting of labeled **nodes** (for example, the nodes are labeled S, NP, and so on in Figure 3.1) connected by **links**. They are called trees because they resemble upside-down trees, and much of the terminology is derived from this analogy with actual trees. The node at the top is called the **root** of the tree, while

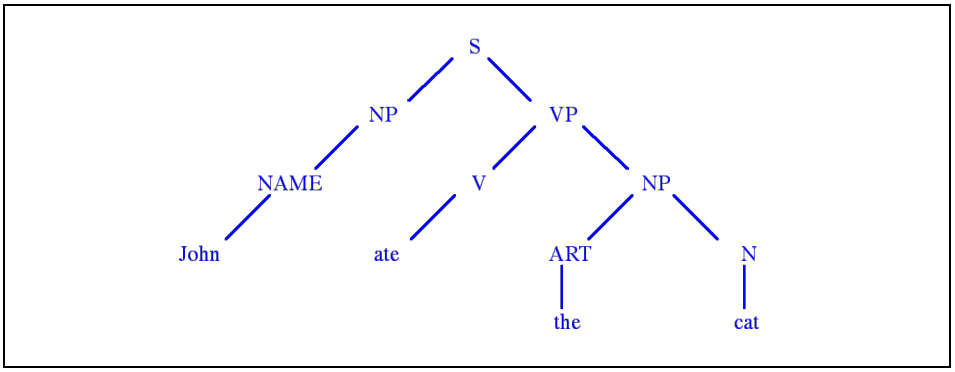


Figure 3.1 A tree representation of *John ate the cat*

1.	S → NP VP	5.	NAME → John
2.	VP → V NP	6.	V → ate
3.	NP → NAME	7.	ART → the
4.	NP → ART N	8.	N → cat

Grammar 3.2 A simple grammar

the nodes at the bottom are called the **leaves**. We say a link points from a **parent** node to a **child** node**.** The node labeled S in Figure 3.1 is the parent node of the nodes labeled NP and VP, and the node labeled NP is in turn the parent node of the node labeled NAME. While every child node has a unique parent, a parent may point to many child nodes. An **ancestor** of a node N is defined as N's parent, or the parent of its parent, and so on. A node is **dominated** by its ancestor nodes. The root node dominates all other nodes in the tree.

To construct a tree structure for a sentence, you must know what structures are legal for English. A set of **rewrite rules** describes what tree structures are allowable. These rules say that a certain symbol may be expanded in the tree by a sequence of other symbols. A set of rules that would allow the tree structure in Figure 3.1 is shown as Grammar 3.2. Rule 1 says that an S may consist of an NP followed by a VP. Rule 2 says that a VP may consist of a V followed by an NP. Rules 3 and 4 say that an NP may consist of a NAME or may consist of an ART followed by an N. Rules 5–8 define possible words for the categories. Grammars consisting entirely of rules with a single symbol on the left-hand side, called the **mother**, are called **context-free grammars** (CFGs). CFGs are a very important class of grammars for two reasons: The formalism is powerful enough to describe most of the structure in natural languages, yet it is restricted enough so that efficient parsers can be built to analyze sentences. Symbols that cannot be further decomposed in a grammar, namely the words in the preceding example, are called **terminal symbols**. The other symbols, such as NP, VP, and S, are called **nonterminal symbols**. The grammatical symbols such as N and V that

describe word categories are called **lexical symbols**. Of course, many words will be listed under multiple categories. For example, *can* would be listed under V and N.

Grammars have a special symbol called the start symbol. In this book, the start symbol will always be S. A grammar is said to **derive** a sentence if there is a sequence of rules that allow you to rewrite the start symbol into the sentence. For instance, Grammar 3.2 derives the sentence *John ate the cat*. This can be seen by showing the sequence of rewrites starting from the S symbol, as follows:

S	
⇒ NP VP	(rewriting S)
⇒ NAME VP	(rewriting NP)
⇒ John VP	(rewriting NAME)
⇒ John V NP	(rewriting VP)
⇒ John ate NP	(rewriting V)
⇒ John ate ART N	(rewriting NP)
⇒ John ate the N	(rewriting ART)
⇒ John ate the cat	(rewriting N)

Two important processes are based on derivations. The first is **sentence generation**, which uses derivations to construct legal sentences. A simple generator could be implemented by randomly choosing rewrite rules, starting from the S symbol, until you have a sequence of words. The preceding example shows that the sentence *John ate the cat* can be generated from the grammar. The second process based on derivations is **parsing**, which identifies the structure of sentences given a grammar. There are two basic methods of searching. A **top-down strategy** starts with the S symbol and then searches through different ways to rewrite the symbols until the input sentence is generated, or until all possibilities have been explored. The preceding example demonstrates that *John ate the cat* is a legal sentence by showing the derivation that could be found by this process.

In a **bottom-up strategy**, you start with the words in the sentence and use the rewrite rules backward to reduce the sequence of symbols until it consists solely of S. The left-hand side of each rule is used to rewrite the symbol on the right-hand side. A possible bottom-up parse of the sentence *John ate the cat* is

⇒ NAME ate the cat	(rewriting John)
⇒ NAME V the cat	(rewriting ate)
⇒ NAME V ART cat	(rewriting the)
⇒ NAME V ART N	(rewriting cat)
⇒ NP V ART N	(rewriting NAME)
⇒ NP V NP	(rewriting ART N)
⇒ NP VP	(rewriting V NP)
⇒ S	(rewriting NP VP)

A tree representation, such as Figure 3.1, can be viewed as a record of the CFG rules that account for the structure of the sentence. In other words, if you kept a record of the parsing process, working either top-down or bottom-up, it would be something similar to the parse tree representation.

3.2 What Makes a Good Grammar

In constructing a grammar for a language, you are interested in **generality**, the range of sentences the grammar analyzes correctly; **selectivity**, the range of non-sentences it identifies as problematic; and **understandability**, the simplicity of the grammar itself.

In small grammars, such as those that describe only a few types of sentences, one structural analysis of a sentence may appear as understandable as another, and little can be said as to why one is superior to the other. As you attempt to extend a grammar to cover a wide range of sentences, however, you often find that one analysis is easily extendable while the other requires complex modification. The analysis that retains its simplicity and generality as it is extended is more desirable.

Unfortunately, here you will be working mostly with small grammars and so will have only a few opportunities to evaluate an analysis as it is extended. You can attempt to make your solutions generalizable, however, by keeping in mind certain properties that any solution should have. In particular, pay close attention to the way the sentence is divided into its subparts, called **constituents**. Besides using your intuition, you can apply a few specific tests, discussed here.

Anytime you decide that a group of words forms a particular constituent, try to construct a new sentence that involves that group of words in a conjunction with another group of words classified as the same type of constituent. This is a good test because for the most part only constituents of the same type can be conjoined. The sentences in Figure 3.3, for example, are acceptable, but the following sentences are not:

*I ate a hamburger and on the stove.
*I ate a cold hot dog and well burned.
*I ate the hot dog slowly and a hamburger.

To summarize, if the proposed constituent doesn't conjoin in some sentence with a constituent of the same class, it is probably incorrect.

Another test involves inserting the proposed constituent into other sentences that take the same category of constituent. For example, if you say that *John's hitting of Mary* is an NP in *John's hitting of Mary alarmed Sue,* then it should be usable as an NP in other sentences as well. In fact this is true—the NP can be the object of a verb, as in *I cannot explain John's hitting of Mary* as well as in the passive form of the initial sentence *Sue was alarmed by John's hitting of Mary*. Given this evidence, you can conclude that the proposed constituent appears to behave just like other NPs.

NP-NP: I ate *a hamburger* and *a hot dog*.
VP-VP: I will *eat the hamburger* and *throw away the hot dog*.
S-S: *I ate a hamburger* and *John ate a hot dog*.
PP-PP: I saw a hot dog *in the bag* and *on the stove*.
ADJP-ADJP: I ate a *cold* and *well burned* hot dog.
ADVP-ADVP: I ate the hot dog *slowly* and *very carefully*.
N-N: I ate a *hamburger* and *hot dog*.
V-V: I will *cook* and *burn* a hamburger.
AUX-AUX: I *can* and *will* eat the hot dog.
ADJ-ADJ: I ate the very *cold* and *burned* hot dog (that is, very cold and very burned).

Figure 3.3 Various forms of conjunctions

As another example of applying these principles, consider the two sentences *I looked up John's phone number* and *I looked up John's chimney*. Should these sentences have the identical structure? If so, you would presumably analyze both as subject-verb-complement sentences with the complement in both cases being a PP. That is, *up John's phone number* would be a PP.

When you try the conjunction test, you should become suspicious of this analysis. Conjoining *up John's phone number* with another PP, as in **I looked up John's phone number and in his cupboards*, is certainly bizarre. Note that *I looked up John's chimney and in his cupboards* is perfectly acceptable. Thus apparently the analysis of *up John's phone number* as a PP is incorrect.

Further evidence against the PP analysis is that *up John's phone number* does not seem usable as a PP in any sentences other than ones involving a few verbs such as *look* or *thought*. Even with the verb *look,* an alternative sentence such as **Up John's phone number, I looked* is quite implausible compared to *Up John's chimney, I looked*.

This type of test can be taken further by considering changing the PP in a manner that usually is allowed. In particular, you should be able to replace the NP *John's phone number* by the pronoun *it*. But the resulting sentence, *I looked up it,* could not be used with the same meaning as *I looked up John's phone number*. In fact, the only way to use a pronoun and retain the original meaning is to use *I looked it up,* corresponding to the form *I looked John's phone number up*.

Thus a different analysis is needed for each of the two sentences. If *up John's phone number* is not a PP, then two remaining analyses may be possible. The VP could be the complex verb *looked up* followed by an NP, or it could consist of three components: the V *looked,* a **particle** *up,* and an NP. Either of these is a better solution. What types of tests might you do to decide between them?

As you develop a grammar, each constituent is used in more and more different ways. As a result, you have a growing number of tests that can be per-

BOX 3.1 Generative Capacity

Grammatical formalisms based on rewrite rules can be compared according to their **generative capacity**, which is the range of languages that each formalism can describe. This book is concerned with natural languages, but it turns out that no natural language can be characterized precisely enough to define generative capacity. Formal languages, however, allow a precise mathematical characterization.

Consider a formal language consisting of the symbols *a, b, c,* and *d* (think of these as words). Then consider a language L1 that allows any sequence of letters in alphabetical order. For example, *abd, ad, bcd, b,* and *abcd* are all legal sentences. To describe this language, we can write a grammar in which the right-hand side of every rule consists of one terminal symbol possibly followed by one nonterminal. Such a grammar is called a **regular** grammar. For L1 the grammar would be

S → a S1	S → d	S1 → d	S3 → d
S → b S2	S1 → b S2	S2 → c S3	
S → c S3	S1 → c S3	S2 → d	

Consider another language, L2, that consists only of sentences that have a sequence of *a*'s followed by an equal number of *b*'s—that is, *ab, aabb, aaabbb,* and so on. You cannot write a regular grammar that can generate L2 exactly. A context-free grammar to generate L2, however, is simple:

S → a b S → a S b

Some languages cannot be generated by a CFG. One example is the language that consists of a sequence of *a*'s, followed by the same number of *b*'s, followed by the same number of *c*'s—that is, *abc, aabbcc, aaabbbccc,* and so on. Similarly, no context-free grammar can generate the language that consists of any sequence of letters repeated in the same order twice, such as *abab, abcabc, acdabacdab,* and so on. There are more general grammatical systems that can generate such sequences, however. One important class is the **context-sensitive grammar**, which consists of rules of the form

$\alpha A \beta \rightarrow \alpha \psi \beta$

where A is a symbol, α and β are (possibly empty) sequences of symbols, and ψ is a nonempty sequence of symbols. Even more general are the **type 0** grammars, which allow arbitrary rewrite rules.

Work in formal language theory began with Chomsky (1956). Since the languages generated by regular grammars are a subset of those generated by context-free grammars, which in turn are a subset of those generated by context-sensitive grammars, which in turn are a subset of those generated by type 0 languages, they form a hierarchy of languages (called the **Chomsky Hierarchy**).

formed to see if a new analysis is reasonable or not. Sometimes the analysis of a new form might force you to back up and modify the existing grammar. This backward step is unavoidable given the current state of linguistic knowledge. The important point to remember, though, is that when a new rule is proposed for a grammar, you must carefully consider its interaction with existing rules.

1.	S → NP VP	4.	VP → V
2.	NP → ART N	5.	VP → V NP
3.	NP → ART ADJ N		

Grammar 3.4

3.3 A Top-Down Parser

A parsing algorithm can be described as a procedure that searches through various ways of combining grammatical rules to find a combination that generates a tree that could be the structure of the input sentence. To keep this initial formulation simple, we will not explicitly construct the tree. Rather, the algorithm will simply return a yes or no answer as to whether such a tree could be built. In other words, the algorithm will say whether a certain sentence is accepted by the grammar or not. This section considers a simple top-down parsing method in some detail and then relates this to work in artificial intelligence (AI) on search procedures.

A top-down parser starts with the S symbol and attempts to rewrite it into a sequence of terminal symbols that matches the classes of the words in the input sentence. The state of the parse at any given time can be represented as a list of symbols that are the result of operations applied so far, called the **symbol list.** For example, the parser starts in the state (S) and after applying the rule S → NP VP the symbol list will be (NP VP). If it then applies the rule NP → ART N, the symbol list will be (ART N VP), and so on.

The parser could continue in this fashion until the state consisted entirely of terminal symbols, and then it could check the input sentence to see if it matched. But this would be quite wasteful, for a mistake made early on (say, in choosing the rule that rewrites S) is not discovered until much later. A better algorithm checks the input as soon as it can. In addition, rather than having a separate rule to indicate the possible syntactic categories for each word, a structure called the **lexicon** is used to efficiently store the possible categories for each word. For now the lexicon will be very simple. A very small lexicon for use in the examples is

cried: V
dogs: N, V
the: ART

With a lexicon specified, a grammar, such as that shown as Grammar 3.4, need not contain any lexical rules.

Given these changes, a state of the parse is now defined by a pair: a symbol list similar to before and a number indicating the current position in the sentence.

Positions fall between the words, with 1 being the position before the first word. For example, here is a sentence with its positions indicated:

1 The 2 dogs 3 cried 4

A typical parse state would be

((N VP) 2)

indicating that the parser needs to find an N followed by a VP, starting at position two. New states are generated from old states depending on whether the first symbol is a lexical symbol or not. If it is a lexical symbol, like N in the preceding example, and if the next word can belong to that lexical category, then you can update the state by removing the first symbol and updating the position counter. In this case, since the word *dogs* is listed as an N in the lexicon, the next parser state would be

((VP) 3)

which means it needs to find a VP starting at position 3. If the first symbol is a nonterminal, like VP, then it is rewritten using a rule from the grammar. For example, using rule 4 in Grammar 3.4, the new state would be

((V) 3)

which means it needs to find a V starting at position 3. On the other hand, using rule 5, the new state would be

((V NP) 3)

A parsing algorithm that is guaranteed to find a parse if there is one must systematically explore every possible new state. One simple technique for this is called **backtracking**. Using this approach, rather than generating a single new state from the state ((VP) 3), you generate all possible new states. One of these is picked to be the next state and the rest are saved as backup states. If you ever reach a situation where the current state cannot lead to a solution, you simply pick a new current state from the list of backup states. Here is the algorithm in a little more detail.

A Simple Top-Down Parsing Algorithm

The algorithm manipulates a list of possible states, called the **possibilities list**. The first element of this list is the **current state**, which consists of a symbol list and a word position in the sentence, and the remaining elements of the search state are the **backup states**, each indicating an alternate symbol-list–word-position pair. For example, the possibilities list

(((N) 2) ((NAME) 1) ((ADJ N) 1))

Step	Current State	Backup States	Comment
1.	((S) 1)		initial position
2.	((NP VP) 1)		rewriting S by rule 1
3.	((ART N VP) 1)	((ART ADJ N VP) 1)	rewriting NP by rules 2 & 3
4.	((N VP) 2)	((ART ADJ N VP) 1)	matching ART with *the*
5.	((VP) 3)	((ART ADJ N VP) 1)	matching N with *dogs*
6.	((V) 3)	((V NP) 3) ((ART ADJ N VP) 1)	rewriting VP by rules 5–8
7.			the parse succeeds as V is matched to *cried*, leaving an empty grammatical symbol list with an empty sentence

Figure 3.5 Top-down depth-first parse of $_1$ *The* $_2$ *dogs* $_3$ *cried* $_4$

indicates that the current state consists of the symbol list (N) at position 2, and that there are two possible backup states: one consisting of the symbol list (NAME) at position 1 and the other consisting of the symbol list (ADJ N) at position 1.

The algorithm starts with the initial state ((S) 1) and no backup states.

1. Select the current state: Take the first state off the possibilities list and call it C. If the possibilities list is empty, then the algorithm fails (that is, no successful parse is possible).
2. If C consists of an empty symbol list and the word position is at the end of the sentence, then the algorithm succeeds.
3. Otherwise, generate the next possible states.
   - 3.1. If the first symbol on the symbol list of C is a lexical symbol, and the next word in the sentence can be in that class, then create a new state by removing the first symbol from the symbol list and updating the word position, and add it to the possibilities list.
   - 3.2. Otherwise, if the first symbol on the symbol list of C is a non-terminal, generate a new state for each rule in the grammar that can rewrite that nonterminal symbol and add them all to the possibilities list.

Consider an example. Using Grammar 3.4, Figure 3.5 shows a trace of the algorithm on the sentence *The dogs cried*. First, the initial S symbol is rewritten using rule 1 to produce a new current state of ((NP VP) 1) in step 2. The NP is

then rewritten in turn, but since there are two possible rules for NP in the grammar, two possible states are generated: The new current state involves (ART N VP) at position 1, whereas the backup state involves (ART ADJ N VP) at position 1. In step 4 a word in category ART is found at position 1 of the sentence, and the new current state becomes (N VP). The backup state generated in step 3 remains untouched. The parse continues in this fashion to step 5, where two different rules can rewrite VP. The first rule generates the new current state, while the other rule is pushed onto the stack of backup states. The parse completes successfully in step 7, since the current state is empty and all the words in the input sentence have been accounted for.

Consider the same algorithm and grammar operating on the sentence

$_1$ The $_2$ old $_3$ man $_4$ cried $_5$

In this case assume that the word *old* is ambiguous between an ADJ and an N and that the word *man* is ambiguous between an N and a V (as in the sentence *The sailors man the boats*). Specifically, the lexicon is

the: ART
old: ADJ, N
man: N, V
cried: V

The parse proceeds as follows (see Figure 3.6). The initial S symbol is rewritten by rule 1 to produce the new current state of ((NP VP) 1). The NP is rewritten in turn, giving the new state of ((ART N VP) 1) with a backup state of ((ART ADJ N VP) 1). The parse continues, finding *the* as an ART to produce the state ((N VP) 2) and then *old* as an N to obtain the state ((VP) 3). There are now two ways to rewrite the VP, giving us a current state of ((V) 3) and the backup states of ((V NP) 3) and ((ART ADJ N) 1) from before. The word *man* can be parsed as a V, giving the state (( ) 4). Unfortunately, while the symbol list is empty, the word position is not at the end of the sentence, so no new state can be generated and a backup state must be used. In the next cycle, step 8, ((V NP) 3) is attempted. Again *man* is taken as a V and the new state ((NP) 4) generated. None of the rewrites of NP yield a successful parse. Finally, in step 12, the last backup state, ((ART ADJ N VP) 1), is tried and leads to a successful parse.

Parsing as a Search Procedure

You can think of parsing as a special case of a **search problem** as defined in AI. In particular, the top-down parser in this section was described in terms of the following generalized search procedure. The possibilities list is initially set to the start state of the parse. Then you repeat the following steps until you have success or failure:

1. Select the first state from the possibilities list (and remove it from the list).

2. Generate the new states by trying every possible option from the selected state (there may be none if we are on a bad path).
3. Add the states generated in step 2 to the possibilities list.

For a **depth-first strategy**, the possibilities list is a stack. In other words, step 1 always takes the first element off the list, and step 3 always puts the new states on the front of the list, yielding a **last-in first-out** (LIFO) strategy.

Step	Current tate	Backup States	Comment
1.	((S) 1)		
2.	((NP VP) 1)		S rewritten to NP VP
3.	((ART N VP) 1)	((ART ADJ N VP) 1)	NP rewritten producing two new states
4.	((N VP) 2)	((ART ADJ N VP) 1)	
5.	((VP) 3)	((ART ADJ N VP) 1)	the backup state remains
6.	((V) 3)	((V NP) 3) ((ART ADJ N VP) 1)	
7.	(( ) 4)	((V NP) 3) ((ART ADJ N VP) 1)	
8.	((V NP) 3)	((ART ADJ N VP) 1)	the first backup is chosen
9.	((NP) 4)	((ART ADJ N VP) 1)	
10.	((ART N) 4)	((ART ADJ N) 4) ((ART ADJ N VP) 1)	looking for ART at 4 fails
11.	((ART ADJ N) 4)	((ART ADJ N VP) 1)	fails again
12.	((ART ADJ N VP) 1)		now exploring backup state saved in step 3
13.	((ADJ N VP) 2)		
14.	((N VP) 3)		
15.	((VP) 4)		
16.	((V) 4)	((V NP) 4)	
17.	(( ) 5)		success!

Figure 3.6 A top-down parse of $_1$ *The* $_2$ *old* $_3$ *man* $_4$ *cried* $_5$

In contrast, in a **breadth-first strategy** the possibilities list is manipulated as a queue. Step 3 adds the new positions onto the end of the list, rather than the beginning, yielding a **first-in first-out** (FIFO) strategy.

We can compare these search strategies using a tree format, as in Figure 3.7, which shows the entire space of parser states for the last example. Each node

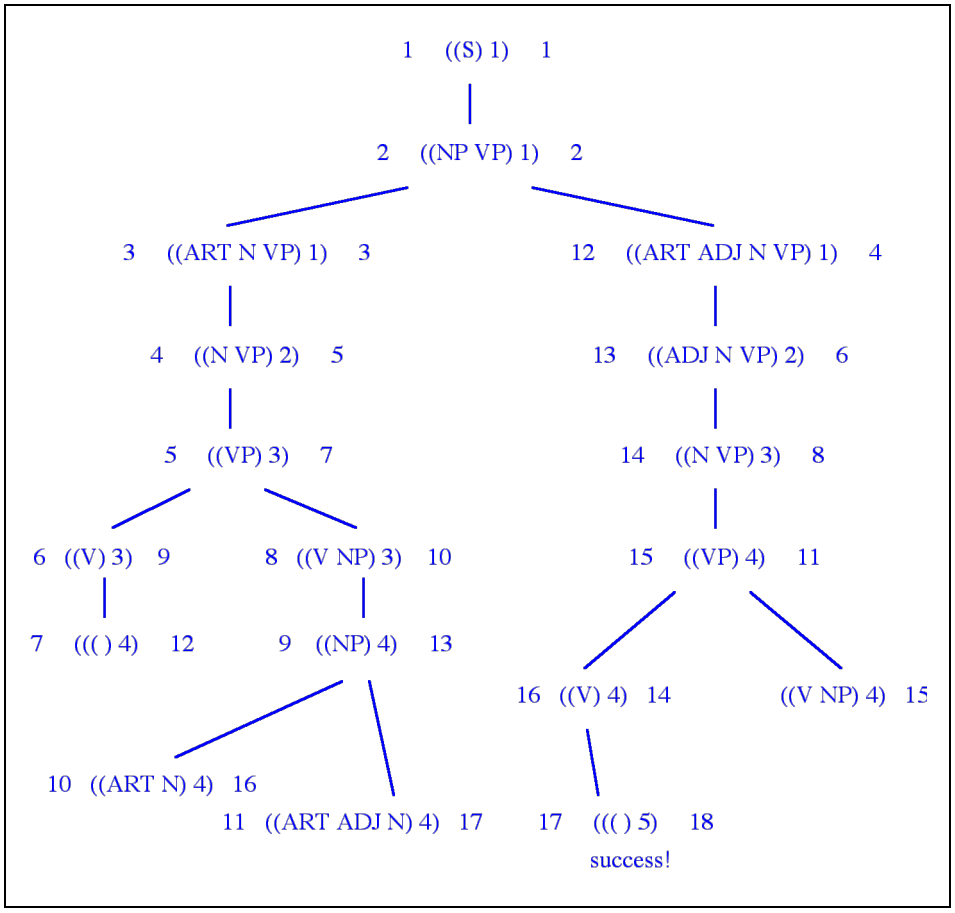


Figure 3.7 Search tree for two parse strategies (depth-first strategy on left; breadth-first on right)

in the tree represents a parser state, and the sons of a node are the possible moves from that state. The number beside each node records when the node was selected to be processed by the algorithm. On the left side is the order produced by the depth-first strategy, and on the right side is the order produced by the breadth-first strategy. Remember, the sentence being parsed is

$_1$ The $_2$ old $_3$ man $_4$ cried $_5$

The main difference between depth-first and breadth-first searches in this simple example is the order in which the two possible interpretations of the first NP are examined. With the depth-first strategy, one interpretation is considered and expanded until it fails; only then is the second one considered. With the breadth-first strategy, both interpretations are considered alternately, each being expanded one step at a time. In this example, both depth-first and breadth-first searches found the solution but searched the space in a different order. A depth-

first search often moves quickly to a solution but in other cases may spend considerable time pursuing futile paths. The breadth-first strategy explores each possible solution to a certain depth before moving on. In this particular example the depth-first strategy found the solution in one less step than the breadth-first. (The state in the bottom right-hand side of Figure 3.7 was not explored by the depth-first parse.)

In certain cases it is possible to put these simple search strategies into an infinite loop. For example, consider a left-recursive rule that could be a first account of the possessive in English (as in the NP *the man's coat*):

NP → NP 's N

With a naive depth-first strategy, a state starting with the nonterminal NP would be rewritten to a new state beginning with NP 's N. But this state also begins with an NP that could be rewritten in the same way. Unless an explicit check were incorporated into the parser, it would rewrite NPs forever! The breadth-first strategy does better with left-recursive rules, as it tries all other ways to rewrite the original NP before coming to the newly generated state with the new NP. But with an ungrammatical sentence it would not terminate because it would rewrite the NP forever while searching for a solution. For this reason, many systems prohibit left-recursive rules from the grammar.

Many parsers built today use the depth-first strategy because it tends to minimize the number of backup states needed and thus uses less memory and requires less bookkeeping.

3.4 A Bottom-Up Chart Parser

The main difference between top-down and bottom-up parsers is the way the grammar rules are used. For example, consider the rule

NP → ART ADJ N

In a top-down system you use the rule to find an NP by looking for the sequence ART ADJ N. In a bottom-up parser you use the rule to take a sequence ART ADJ N that you have found and identify it as an NP. The basic operation in bottom-up parsing then is to take a sequence of symbols and match it to the right-hand side of the rules. You could build a bottom-up parser simply by formulating this matching process as a search process. The state would simply consist of a symbol list, starting with the words in the sentence. Successor states could be generated by exploring all possible ways to

- rewrite a word by its possible lexical categories
- replace a sequence of symbols that matches the right-hand side of a grammar rule by its left-hand side symbol

Unfortunately, such a simple implementation would be prohibitively expensive, as the parser would tend to try the same matches again and again, thus dupli-

1. S → NP VP
2. NP → ART ADJ N
3. NP → ART N
4. NP → ADJ N
5. VP → AUX VP
6. VP → V NP

Grammar 3.8 A simple context-free grammar

cating much of its work unnecessarily. To avoid this problem, a data structure called a **chart** is introduced that allows the parser to store the partial results of the matching it has done so far so that the work need not be reduplicated.

Matches are always considered from the point of view of one constituent, called the **key**. To find rules that match a string involving the key, look for rules that start with the key, or for rules that have already been started by earlier keys and require the present key either to complete the rule or to extend the rule. For instance, consider Grammar 3.8.

Assume you are parsing a sentence that starts with an ART. With this ART as the key, rules 2 and 3 are matched because they start with ART. To record this for analyzing the next key, you need to record that rules 2 and 3 could be continued at the point after the ART. You denote this fact by writing the rule with a dot (◦), indicating what has been seen so far. Thus you record

2′. NP → ART ◦ ADJ N
3′. NP → ART ◦ N

If the next input key is an ADJ, then rule 4 may be started, and the modified rule 2′ may be extended to give

2′′. NP → ART ADJ ◦ N

The chart maintains the record of all the constituents derived from the sentence so far in the parse. It also maintains the record of rules that have matched partially but are not complete. These are called the **active arcs**. For example, after seeing an initial ART followed by an ADJ in the preceding example, you would have the chart shown in Figure 3.9. You should interpret this figure as follows. There are two completed constituents on the chart: ART1 from position 1 to 2 and ADJ1 from position 2 to 3. There are four active arcs indicating possible constituents. These are indicated by the arrows and are interpreted as follows (from top to bottom). There is a potential NP starting at position 1, which needs an ADJ starting at position 2. There is another potential NP starting at position 1, which needs an N starting at position 2. There is a potential NP starting at position 2 with an ADJ, which needs an N starting at position 3. Finally, there is a potential NP starting at position 1 with an ART and then an ADJ, which needs an N starting at position 3.

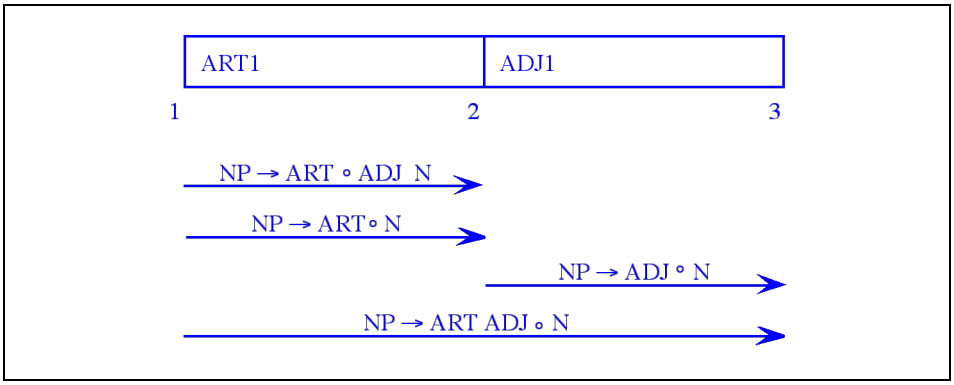


Figure 3.9 The chart after seeing an ADJ in position 2

To add a constituent C from position p_1 to p_2:

1. Insert C into the chart from position p_1 to p_2.
2. For any active arc of the form $X \rightarrow X_1 \ldots \circ C \ldots X_n$ from position p_0 to p_1, add a new active arc $X \rightarrow X_1 \ldots C \circ \ldots X_n$ from position p_0 to p_2.
3. For any active arc of the form $X \rightarrow X_1 \ldots X_n \circ C$ from position p_0 to p_1, then add a new constituent of type X from p_0 to p_2 to the agenda.

Figure 3.10 The arc extension algorithm

The basic operation of a chart-based parser involves combining an active arc with a completed constituent. The result is either a new completed constituent or a new active arc that is an extension of the original active arc. New completed constituents are maintained on a list called the **agenda** until they themselves are added to the chart. This process is defined more precisely by the arc extension algorithm shown in Figure 3.10. Given this algorithm, the bottom-up chart parsing algorithm is specified in Figure 3.11.

As with the top-down parsers, you may use a depth-first or breadth-first search strategy, depending on whether the agenda is implemented as a stack or a queue. Also, for a full breadth-first strategy, you would need to read in the entire input and add the interpretations of the words onto the agenda before starting the algorithm. Let us assume a depth-first search strategy for the following example.

Consider using the algorithm on the sentence *The large can can hold the water* using Grammar 3.8 with the following lexicon:

the: ART
large: ADJ
can: N, AUX, V
hold: N, V
water: N, V

Do until there is no input left:

1. If the agenda is empty, look up the interpretations for the next word in the input and add them to the agenda.
2. Select a constituent from the agenda (let's call it constituent C from position p_1 to p_2).
3. For each rule in the grammar of form $X \rightarrow C\ X_1 \ldots X_n$, add an active arc of form $X \rightarrow \circ\ C\ X_1 \ldots X_n$ from position p_1 to p_2.
4. Add C to the chart using the arc extension algorithm above.

Figure 3.11 A bottom-up chart parsing algorithm

To best understand the example, draw the chart as it is extended at each step of the algorithm. The agenda is initially empty, so the word *the* is read and a constituent ART1 placed on the agenda.

Entering ART1: (*the* from 1 to 2)
Adds active arc NP → ART ∘ ADJ N from 1 to 2
Adds active arc NP → ART ∘ N from 1 to 2

Both these active arcs were added by step 3 of the parsing algorithm and were derived from rules 2 and 3 in the grammar, respectively. Next the word *large* is read and a constituent ADJ1 is created.

Entering ADJ1: (*large* from 2 to 3)
Adds arc NP → ADJ ∘ N from 2 to 3
Adds arc NP → ART ADJ ∘ N from 1 to 3

The first arc was added in step 3 of the algorithm. The second arc added here is an extension of the first active arc that was added when ART1 was added to the chart using the arc extension algorithm (step 4).

The chart at this point has already been shown in Figure 3.9. Notice that active arcs are never removed from the chart. For example, when the arc NP → ART ∘ ADJ N from 1 to 2 was extended, producing the arc from 1 to 3, both arcs remained on the chart. This is necessary because the arcs could be used again in a different way by another interpretation.

For the next word, *can*, three constituents, N1, AUX1, and V1 are created for its three interpretations.

Entering N1: (*can* from 3 to 4)

No active arcs are added in step 2, but two are completed in step 4 by the arc extension algorithm, producing two NPs that are added to the agenda: The first, an NP from 1 to 4, is constructed from rule 2, while the second, an NP from 2 to 4, is constructed from rule 4. These NPs are now at the top of the agenda.

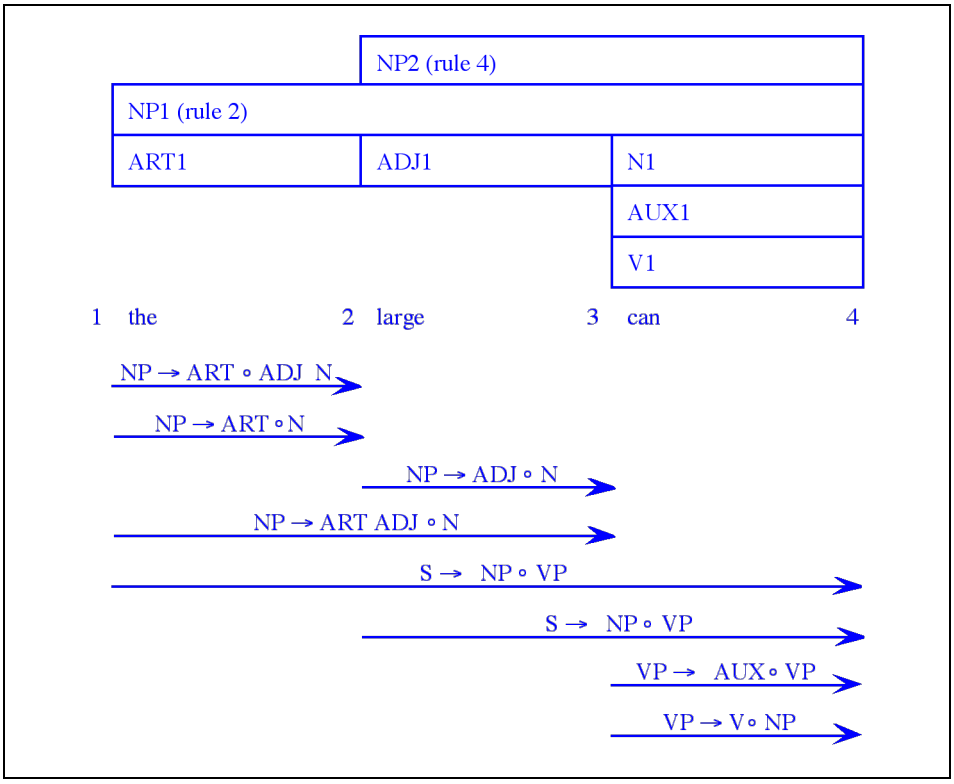


Figure 3.12 After parsing *the large can*

Entering NP1: an NP (*the large can* from 1 to 4)
Adding active arc S → NP ∘ VP from 1 to 4
Entering NP2: an NP (*large can* from 2 to 4)
Adding arc S → NP ∘ VP from 2 to 4
Entering AUX1: (*can* from 3 to 4)
Adding arc VP → AUX ∘ VP from 3 to 4
Entering V1: (*can* from 3 to 4)
Adding arc VP → V ∘ NP from 3 to 4

The chart is shown in Figure 3.12, which illustrates all the completed constituents (NP2, NP1, ART1, ADJ1, N1, AUX1, V1) and all the uncompleted active arcs entered so far. The next word is *can* again, and N2, AUX2, and V2 are created.

Entering N2: (*can* from 4 to 5, the second *can*)
Adds no active arcs
Entering AUX2: (*can* from 4 to 5)
Adds arc VP → AUX ∘ VP from 4 to 5

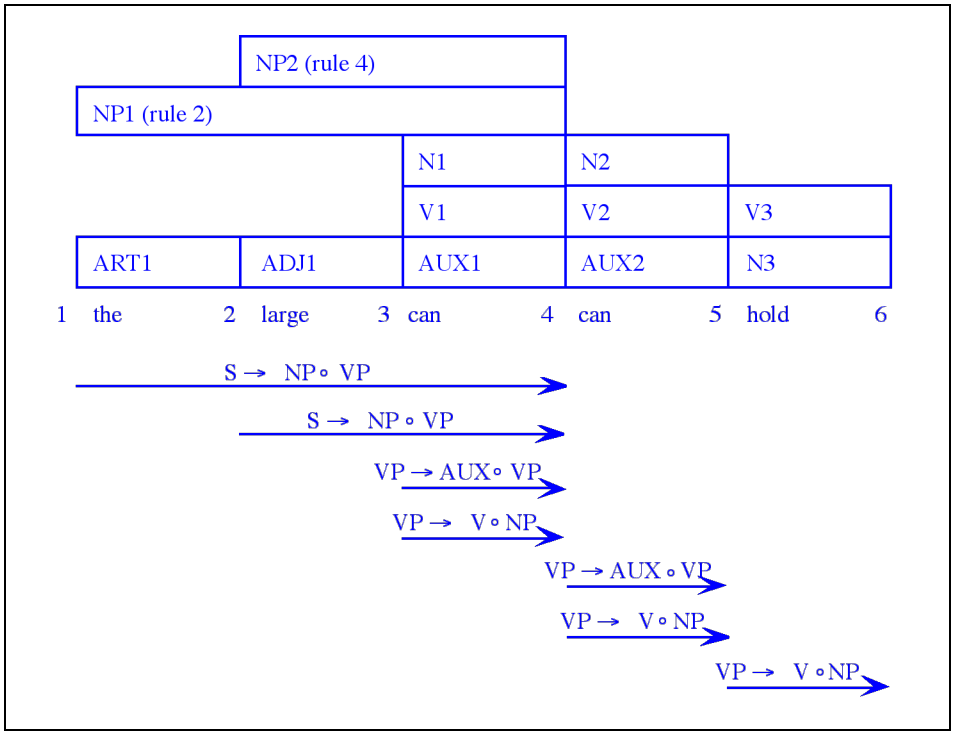


Figure 3.13 The chart after adding *hold*, omitting arcs generated for the first NP

Entering V2: (*can* from 4 to 5)
Adds arc VP → V ∘ NP from 4 to 5

The next word is *hold*, and N3 and V3 are created.

Entering N3: (*hold* from 5 to 6)
Adds no active arcs
Entering V3: (*hold* from 5 to 6)
Adds arc VP → V ∘ NP from 5 to 6

The chart in Figure 3.13 shows all the completed constituents built so far, together with all the active arcs, except for those used in the first NP.

Entering ART2: (*the* from 6 to 7)
Adding arc NP → ART ∘ ADJ N from 6 to 7
Adding arc NP → ART ∘ N from 6 to 7

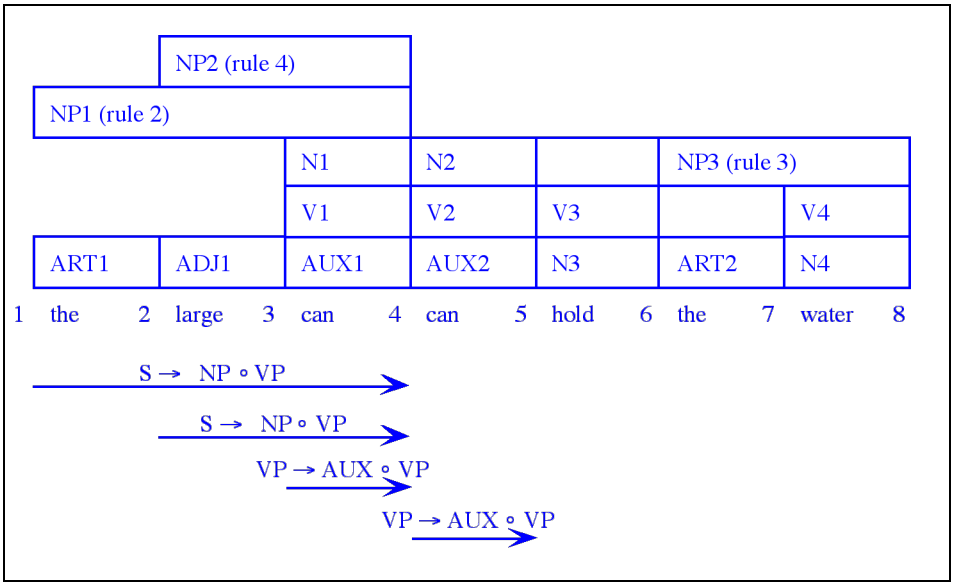


Figure 3.14 The chart after all the NPs are found, omitting all but the crucial active arcs

Entering N4: (*water* from 7 to 8)
No active arcs added in step 3
An NP, NP3, from 6 to 8 is pushed onto the agenda, by completing
arc NP → ART ∘ N from 6 to 7
Entering NP3: (*the water* from 6 to 8)
A VP, VP1, from 5 to 8 is pushed onto the agenda, by completing
VP → V ∘ NP from 5 to 6
Adds arc S → NP ∘ VP from 6 to 8

The chart at this stage is shown in Figure 3.14, but only the active arcs to be used in the remainder of the parse are shown.

Entering VP1: (*hold the water* from 5 to 8)
A VP, VP2, from 4 to 8 is pushed onto the agenda, by completing
VP → AUX ∘ VP from 4 to 5
Entering VP2: (*can hold the water* from 4 to 8)
An S, S1, is added from 1 to 8, by completing
arc S → NP ∘ VP from 1 to 4
A VP, VP3, is added from 3 to 8, by completing
arc VP → AUX ∘ VP from 3 to 4
An S, S2, is added from 2 to 8, by completing
arc S → NP ∘ VP from 2 to 4

Since you have derived an S covering the entire sentence, you can stop successfully. If you wanted to find all possible interpretations for the sentence,

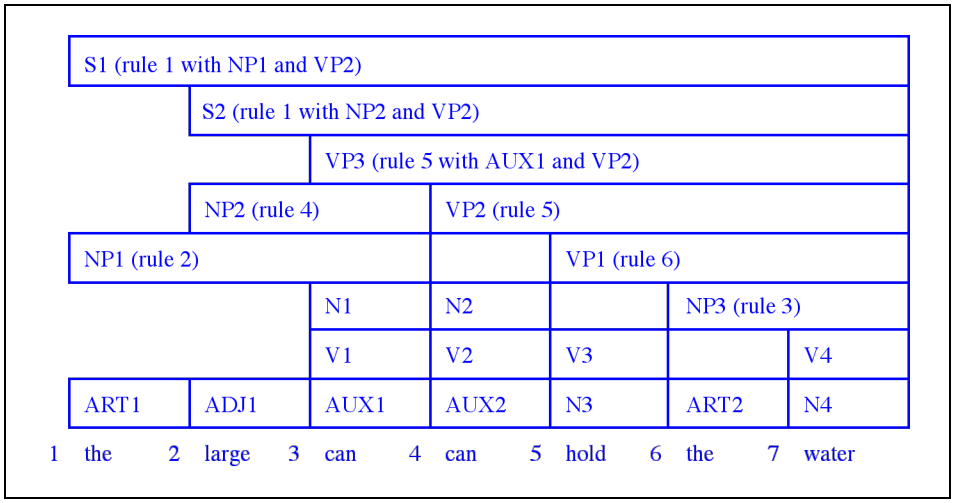


Figure 3.15 The final chart

you would continue parsing until the agenda became empty. The chart would then contain as many S structures covering the entire set of positions as there were different structural interpretations. In addition, this representation of the entire set of structures would be more efficient than a list of interpretations, because the different S structures might share common subparts represented in the chart only once. Figure 3.15 shows the final chart.

Efficiency Considerations

Chart-based parsers can be considerably more efficient than parsers that rely only on a search because the same constituent is never constructed more than once. For instance, a pure top-down or bottom-up search strategy could require up to C^n operations to parse a sentence of length n, where C is a constant that depends on the specific algorithm you use. Even if C is very small, this exponential complexity rapidly makes the algorithm unusable. A chart-based parser, on the other hand, in the worst case would build every possible constituent between every possible pair of positions. This allows us to show that it has a worst-case complexity of $K*n^3$, where n is the length of the sentence and K is a constant depending on the algorithm. Of course, a chart parser involves more work in each step, so K will be larger than C. To contrast the two approaches, assume that C is 10 and that K is a hundred times worse, 1000. Given a sentence of 12 words, the brute force search might take 10^{12} operations (that is, 1,000,000,000,000), whereas the chart parser would take $1000 * 12^3$ (that is, 1,728,000). Under these assumptions, the chart parser would be up to 500,000 times faster than the brute force search on some examples!

3.5 Transition Network Grammars

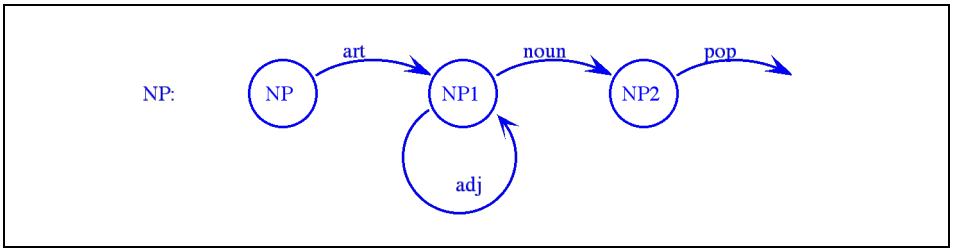
So far we have examined only one formalism for representing grammars, namely context-free rewrite rules. Here we consider another formalism that is useful in a wide range of applications. It is based on the notion of a **transition network** consisting of **nodes** and **labeled arcs**. One of the nodes is specified as the **initial state**, or **start state**. Consider the network named NP in Grammar 3.16, with the initial state labeled NP and each arc labeled with a word category. Starting at the initial state, you can traverse an arc if the current word in the sentence is in the category on the arc. If the arc is followed, the current word is updated to the next word. A phrase is a legal NP if there is a path from the node NP to a **pop arc** (an arc labeled pop) that accounts for every word in the phrase. This network recognizes the same set of sentences as the following context-free grammar:

NP → ART NP1
NP1 → ADJ NP1
NP1 → N

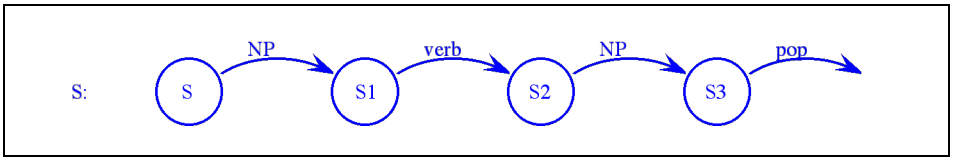
Consider parsing the NP *a purple cow* with this network. Starting at the node NP, you can follow the arc labeled art, since the current word is an article—namely, *a*. From node NP1 you can follow the arc labeled adj using the adjective *purple,* and finally, again from NP1, you can follow the arc labeled noun using the noun *cow*. Since you have reached a pop arc, *a purple cow* is a legal NP.

Simple transition networks are often called **finite state machines** (FSMs). Finite state machines are equivalent in expressive power to regular grammars (see Box 3.2), and thus are not powerful enough to describe all languages that can be described by a CFG. To get the descriptive power of CFGs, you need a notion of recursion in the network grammar. A **recursive transition network** (RTN) is like a simple transition network, except that it allows arc labels to refer to other networks as well as word categories. Thus, given the NP network in Grammar 3.16, a network for simple English sentences can be expressed as shown in Grammar 3.17. Uppercase labels refer to networks. The arc from S to S1 can be followed only if the NP network can be successfully traversed to a pop arc. Although not shown in this example, RTNs allow true recursion—that is, a network might have an arc labeled with its own name.

Consider finding a path through the S network for the sentence *The purple cow ate the grass*. Starting at node S, to follow the arc labeled NP, you need to traverse the NP network. Starting at node NP, traverse the network as before for the input *the purple cow*. Following the pop arc in the NP network, return to the S network and traverse the arc to node S1. From node S1 you follow the arc labeled verb using the word *ate*. Finally, the arc labeled NP can be followed if you can traverse the NP network again. This time the remaining input consists of the words *the grass*. You follow the arc labeled art and then the arc labeled noun in the NP network; then take the pop arc from node NP2 and then another pop from node S3. Since you have traversed the network and used all the words in the sentence, *The purple cow ate the grass* is accepted as a legal sentence.



Grammar 3.16



Grammar 3.17

Arc Type	Example	How Used
CAT	noun	succeeds only if current word is of the named category
WRD	of	succeeds only if current word is identical to the label
PUSH	NP	succeeds only if named network can be successfully traversed
JUMP	jump	always succeeds
POP	pop	succeeds and signals the successful end of the network

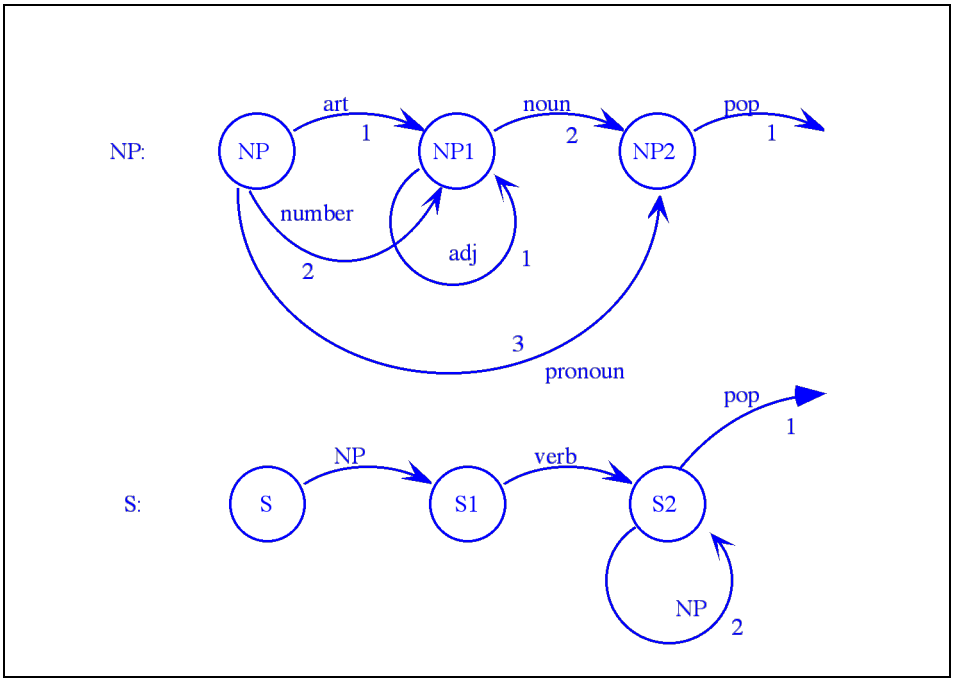
Figure 3.18 The arc labels for RTNs

In practice, RTN systems incorporate some additional arc types that are useful but not formally necessary. Figure 3.18 summarizes the arc types, together with the notation that will be used in this book to indicate these arc types. According to this terminology, arcs that are labeled with networks are called **push arcs**, and arcs labeled with word categories are called **cat arcs**. In addition, an arc that can always be followed is called a **jump arc**.

Top-Down Parsing with Recursive Transition Networks

An algorithm for parsing with RTNs can be developed along the same lines as the algorithms for parsing CFGs. The state of the parse at any moment can be represented by the following:

current position—a pointer to the next word to be parsed.
current node—the node at which you are located in the network.
return points—a stack of nodes in other networks where you will continue if you **pop** from the current network.



Grammar 3.19

First, consider an algorithm for searching an RTN that assumes that if you can follow an arc, it will be the correct one in the final parse. Say you are in the middle of a parse and know the three pieces of information just cited. You can leave the current node and traverse an arc in the following cases:

Case 1: **If** arc names word category and next word in sentence is in that category,
Then (1) update *current position* to start at the next word;
(2) update *current node* to the destination of the arc.

Case 2: **If** arc is a push arc to a network N,
Then (1) add the destination of the arc onto *return points*;
(2) update *current node* to the starting node in network N.

Case 3: **If** arc is a pop arc and *return points* list is not empty,
Then (1) remove first return point and make it *current node*.

Case 4: **If** arc is a pop arc, *return points* list is empty and there are no words left,
Then (1) parse completes successfully.

Grammar 3.19 shows a network grammar. The numbers on the arcs simply indicate the order in which arcs will be tried when more than one arc leaves a node.

Step	Current Node	Current Position	Return Points	Arc to be Followed	Comments
1.	(S,	1,	NIL)	S/1	initial position
2.	(NP,	1,	(S1))	NP/1	followed push arc to NP network, to return ultimately to S1
3.	(NP1,	2,	(S1))	NP1/1	followed arc NP/1 (*the*)
4.	(NP1,	3,	(S1))	NP1/2	followed arc NP1/1 (*old*)
5.	(NP2,	4,	(S1))	NP2/2	followed arc NP1/2 (*man*) since NP1/1 is not applicable
6.	(S1,	4,	NIL)	S1/1	the pop arc gets us back to S1
7.	(S2,	5,	NIL)	S2/1	followed arc S2/1 (*cried*)
8.					parse succeeds on pop arc from S2

Figure 3.20 A trace of a top-down parse

Figure 3.20 demonstrates that the grammar accepts the sentence

$_1$ The $_2$ old $_3$ man $_4$ cried $_5$

by showing the sequence of parse states that can be generated by the algorithm. In the trace, each arc is identified by the name of the node that it leaves plus the number identifier. Thus arc S/1 is the arc labeled 1 leaving the S node. If you start at node S, the only possible arc to follow is the push arc NP. As specified in case 2 of the algorithm, the new parse state is computed by setting the current node to NP and putting node S1 on the return points list. From node NP, arc NP/1 is followed and, as specified in case 1 of the algorithm, the input is checked for a word in category art. Since this check succeeds, the arc is followed and the current position is updated (step 3). The parse continues in this manner to step 5, when a pop arc is followed, causing the current node to be reset to S1 (that is, the NP arc succeeded). The parse succeeds after finding a verb in step 6 and following the pop arc from the S network in step 7.

In this example the parse succeeded because the first arc that succeeded was ultimately the correct one in every case. However, with a sentence like *The green faded,* where *green* can be an adjective or a noun, this algorithm would fail because it would initially classify *green* as an adjective and then not find a noun following. To be able to recover from such failures, we save all possible backup states as we go along, just as we did with the CFG top-down parsing algorithm.

Consider this technique in operation on the following sentence:

$_1$ One $_2$ saw $_3$ the $_4$ man $_5$

The parser initially attempts to parse the sentence as beginning with the NP *one saw,* but after failing to find a verb, it backtracks and finds a successful parse starting with the NP *one*. The trace of the parse is shown in Figure 3.21, where at

Step	Current State	Arc to be Followed	Backup States
1.	(S, 1, NIL)	S/1	NIL
2.	(NP, 1, (S1))	NP/2 (& NP/3 for backup)	NIL
3.	(NP1, 2, (S1))	NP1/2	(NP2, 2, (S1))
4.	(NP2, 3, (S1))	NP2/1	(NP2, 2, (S1))
5.	(S1, 3, NIL)	no arc can be followed	(NP2, 2, (S1))
6.	(NP2, 2, (S1))	NP2/1	NIL
7.	(S1, 2, NIL)	S1/1	NIL
8.	(S2, 3, NIL)	S2/2	NIL
9.	(NP, 3, (S2))	NP/1	NIL
10.	(NP1, 4, (S2))	NP1/2	NIL
11.	(NP2, 5, (S2))	NP2/1	NIL
12.	(S2, 5, NIL)	S2/1	NIL
13.	parse succeeds		NIL

Figure 3.21 A top-down RTN parse with backtracking

each stage the current parse state is shown in the form of a triple (current node, current position, return points), together with possible states for backtracking. The figure also shows the arcs used to generate the new state and backup states.

This trace behaves identically to the previous example except in two places. In step 2, two arcs leaving node NP could accept the word *one*. Arc NP/2 classifies *one* as a number and produces the next current state. Arc NP/3 classifies it as a pronoun and produces a backup state. This backup state is actually used later in step 6 when it is found that none of the arcs leaving node S1 can accept the input word *the*.

Of course, in general, many more backup states are generated than in this simple example. In these cases there will be a list of possible backup states. Depending on how this list is organized, you can produce different orderings on when the states are examined.

An RTN parser can be constructed to use a chart-like structure to gain the advantages of chart parsing. In RTN systems, the chart is often called the **well-formed substring table** (WFST). Each time a pop is followed, the constituent is placed on the WFST, and every time a push is found, the WFST is checked before the subnetwork is invoked. If the chart contains constituent(s) of the type being pushed for, these are used and the subnetwork is not reinvoked. An RTN using a WFST has the same complexity as the chart parser described in the last section: $K*n^3$, where n is the length of the sentence.

◦ 3.6 Top-Down Chart Parsing

So far, you have seen a simple top-down method and a bottom-up chart-based method for parsing context-free grammars. Each of the approaches has its advantages and disadvantages. In this section a new parsing method is presented that

actually captures the advantages of both. But first, consider the pluses and minuses of the approaches.

Top-down methods have the advantage of being highly predictive. A word might be ambiguous in isolation, but if some of those possible categories cannot be used in a legal sentence, then these categories may never even be considered. For example, consider Grammar 3.8 in a top-down parse of the sentence *The can holds the water,* where *can* may be an AUX, V, or N, as before.

The top-down parser would rewrite (S) to (NP VP) and then rewrite the NP to produce three possibilities, (ART ADJ N VP), (ART N VP), and (ADJ N VP). Taking the first, the parser checks if the first word, *the*, can be an ART, and then if the next word, *can*, can be an ADJ, which fails. Trying the next possibility, the parser checks *the* again, and then checks if *can* can be an N, which succeeds. The interpretations of *can* as an auxiliary and a main verb are never considered because no syntactic tree generated by the grammar would ever predict an AUX or V in this position. In contrast, the bottom-up parser would have considered all three interpretations of *can* from the start—that is, all three would be added to the chart and would combine with active arcs. Given this argument, the top-down approach seems more efficient.

On the other hand, consider the top-down parser in the example above needed to check that the word *the* was an ART twice, once for each rule. This reduplication of effort is very common in pure top-down approaches and becomes a serious problem, and large constituents may be rebuilt again and again as they are used in different rules. In contrast, the bottom-up parser only checks the input once, and only builds each constituent exactly once. So by this argument, the bottom-up approach appears more efficient.

You can gain the advantages of both by combining the methods. A small variation in the bottom-up chart algorithm yields a technique that is predictive like the top-down approaches yet avoids any reduplication of work as in the bottom-up approaches.

As before, the algorithm is driven by an agenda of completed constituents and the arc extension algorithm, which combines active arcs with constituents when they are added to the chart. While both use the technique of extending arcs with constituents, the difference is in how new arcs are generated from the grammar. In the bottom-up approach, new active arcs are generated whenever a completed constituent is added that could be the first constituent of the right-hand side of a rule. With the top-down approach, new active arcs are generated whenever a new active arc is added to the chart, as described in the top-down arc introduction algorithm shown in Figure 3.22. The parsing algorithm is then easily stated, as is also shown in Figure 3.22.

Consider this new algorithm operating with the same grammar on the same sentence as in Section 3.4, namely *The large can can hold the water*. In the initialization stage, an arc labeled S → ∘ NP VP is added. Then, active arcs for each rule that can derive an NP are added: NP → ∘ ART ADJ N, NP → ∘ ART N,

Top-Down Arc Introduction Algorithm

To add an arc $S \rightarrow C_1 \ldots \circ C_i \ldots C_n$ ending at position j, do the following:

For each rule in the grammar of form $C_i \rightarrow X_1 \ldots X_k$, recursively add the new arc $C_i \rightarrow \circ X_1 \ldots X_k$ from position j to j.

Top-Down Chart Parsing Algorithm

Initialization: For every rule in the grammar of form $S \rightarrow X_1 \ldots X_k$, add an arc labeled $S \rightarrow \circ X_1 \ldots X_k$ using the arc introduction algorithm.

Parsing: Do until there is no input left:

1. If the agenda is empty, look up the interpretations of the next word and add them to the agenda.
2. Select a constituent from the agenda (call it constituent C).
3. Using the arc extension algorithm, combine C with every active arc on the chart. Any new constituents are added to the agenda.
4. For any active arcs created in step 3, add them to the chart using the top-down arc introduction algorithm.

Figure 3.22 The top-down arc introduction and chart parsing algorithms

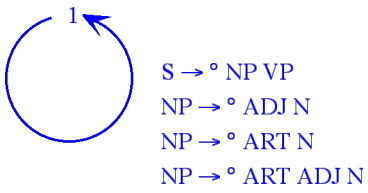


Figure 3.23 The initial chart

and NP → ° ADJ N are all added from position 1 to 1. Thus the initialized chart is as shown in Figure 3.23. The trace of the parse is as follows:

Entering ART1 (*the*) from 1 to 2
  Two arcs can be extended by the arc extension algorithm
    NP → ART ° N from 1 to 2
    NP → ART ° ADJ N from 1 to 2
Entering ADJ1 (*large*) from 2 to 3
  One arc can be extended
    NP → ART ADJ ° N from 1 to 3
Entering AUX1 (*can*) from 3 to 4
  No activity, constituent is ignored
Entering V1 (*can*) from 3 to 4
  No activity, constituent is ignored

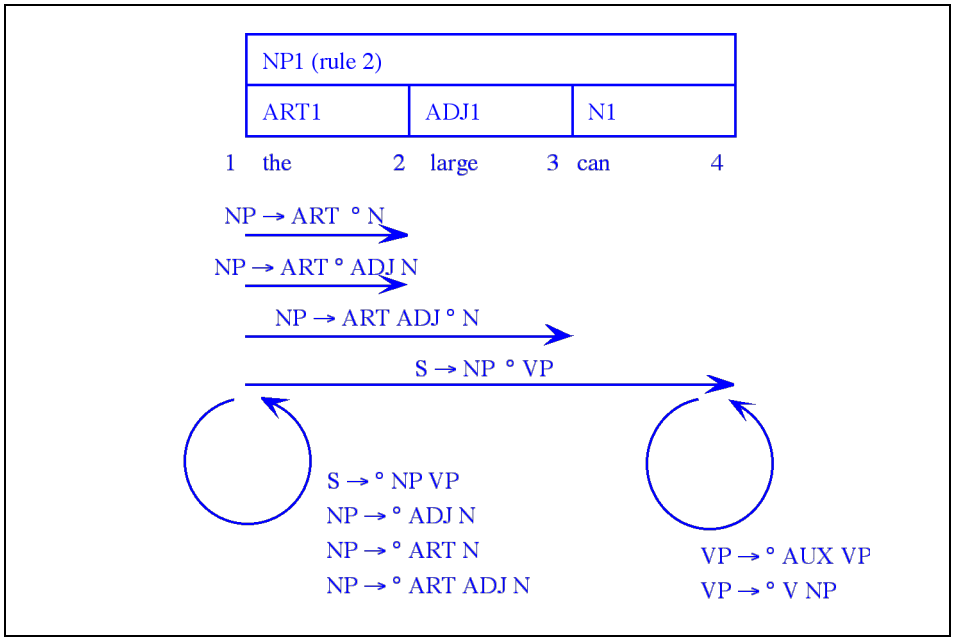


Figure 3.24 The chart after building the first NP

Entering N1 (*can*) from 3 to 4
One arc extended and completed yielding
NP1 from 1 to 4 (*the large can*)
Entering NP1 from 1 to 4
One arc can be extended
S → NP ◦ VP from 1 to 4
Using the top-down rule (step 4), new active arcs are added for VP
VP → ◦ AUX VP from 4 to 4
VP → ◦ V NP from 4 to 4

At this stage, the chart is as shown in Figure 3.24. Compare this with Figure 3.10. It contains fewer completed constituents since only those that are allowed by the top-down filtering have been constructed.

The algorithm continues, adding the three interpretations of *can* as an AUX, V, and N. The AUX reading extends the VP → ◦ AUX VP arc at position 4 and adds active arcs for a new VP starting at position 5. The V reading extends the VP → ◦ V NP arc and adds active arcs for an NP starting at position 5. The N reading does not extend any arc and so is ignored. After the two readings of *hold* (as an N and V) are added, the chart is as shown in Figure 3.25. Again, compare with the corresponding chart for the bottom-up parser in Figure 3.13. The rest of the sentence is parsed similarly, and the final chart is shown in Figure 3.26. In comparing this to the final chart produced by the bottom-up parser (Figure 3.15),

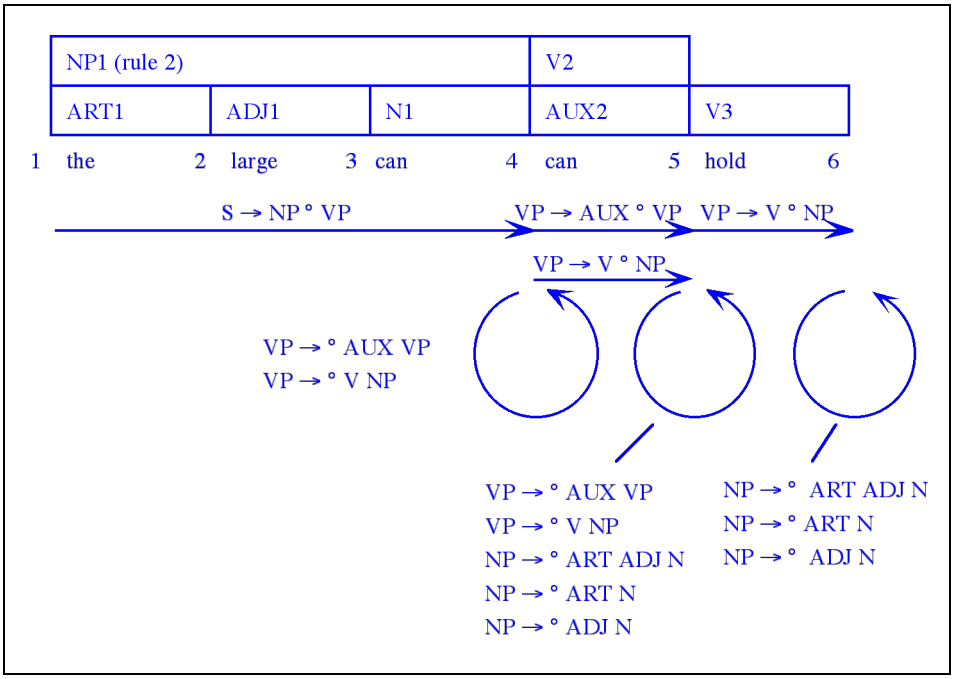


Figure 3.25 The chart after adding *hold*, omitting arcs generated for the first NP

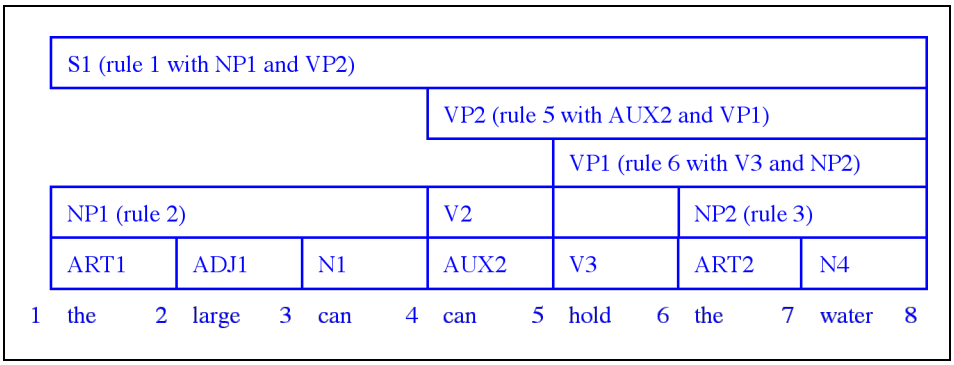


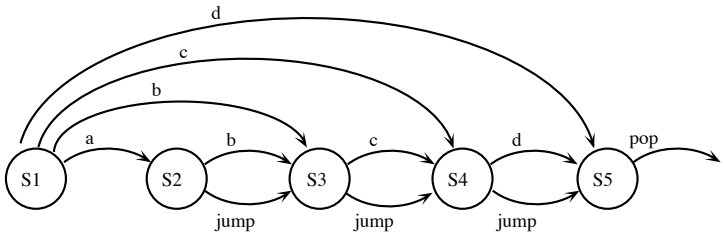
Figure 3.26 The final chart for the top-down filtering algorithm

you see that the number of constituents generated has dropped from 21 to 13. While it is not a big difference here with such a simple grammar, the difference can be dramatic with a sizable grammar.

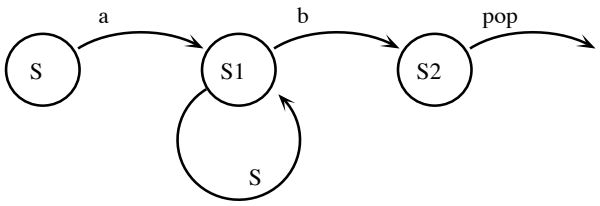
It turns out in the worst-case analysis that the top-down chart parser is not more efficient that the pure bottom-up chart parser. Both have a worst-case complexity of $K*n^3$ for a sentence of length n. In practice, however, the top-down method is considerably more efficient for any reasonable grammar.

BOX 3.2 Generative Capacity of Transition Networks

Transition network systems can be classified by the types of languages they can describe. In fact, you can draw correspondences between various network systems and rewrite-rule systems. For instance, simple transition networks (that is, finite state machines) with no push arcs are expressively equivalent to regular grammars—that is, every language that can be described by a simple transition network can be described by a regular grammar, and vice versa. An FSM for the first language described in Box 3.1 is



Recursive transition networks, on the other hand, are expressively equivalent to context-free grammars. Thus an RTN can be converted into a CFG and vice versa. A recursive transition network for the language consisting of a number of *a*'s followed by an equal number of *b*'s is



3.7 Finite State Models and Morphological Processing

Although in simple examples and small systems you can list all the words allowed by the system, large vocabulary systems face a serious problem in representing the lexicon. Not only are there a large number of words, but each word may combine with affixes to produce additional related words. One way to address this problem is to preprocess the input sentence into a sequence of morphemes. A word may consist of single morpheme, but often a word consists of a root form plus an affix. For instance, the word *eaten* consists of the root form *eat* and the suffix *-en,* which indicates the past participle form. Without any preprocessing, a lexicon would have to list all the forms of *eat,* including *eats, eating, ate,* and *eaten*. With preprocessing, there would be one morpheme *eat* that may combine with suffixes such as *-ing, -s,* and *-en,* and one entry for the irregular form *ate*. Thus the lexicon would only need to store two entries (*eat* and *ate*) rather than four. Likewise the word *happiest* breaks down into the root form

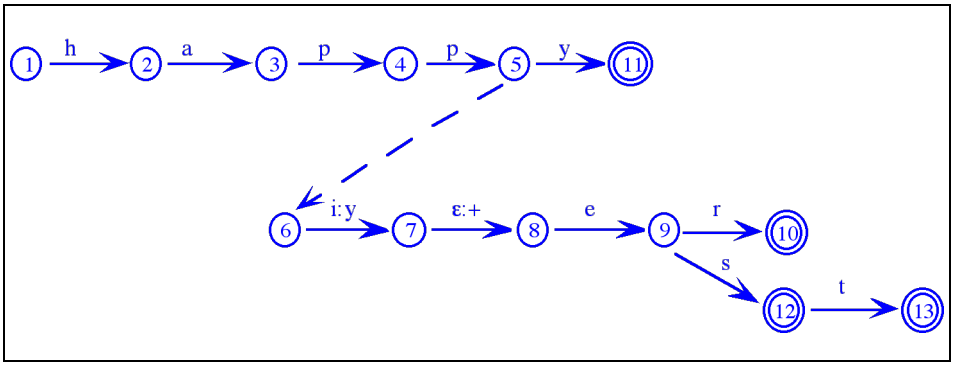


Figure 3.27 A simple FST for the forms of *happy*

happy and the suffix *-est*, and thus does not need a separate entry in the lexicon. Of course, not all forms are allowed; for example, the word *seed* cannot be decomposed into a root form *se* (or *see*) and a suffix *-ed*. The lexicon would have to encode what forms are allowed with each root.

One of the most popular models is based on **finite state transducers (FSTs)**, which are like finite state machines except that they produce an output given an input. An arc in an FST is labeled with a pair of symbols. For example, an arc labeled i:y could only be followed if the current input is the letter *i* and the output is the letter *y*. FSTs can be used to concisely represent the lexicon and to transform the surface form of words into a sequence of morphemes. Figure 3.27 shows a simple FST that defines the forms of the word *happy* and its derived forms. It transforms the word *happier* into the sequence *happy +er* and *happiest* into the sequence *happy +est*.

Arcs labeled by a single letter have that letter as both the input and the output. Nodes that are double circles indicate success states, that is, acceptable words. Consider processing the input word *happier* starting from state 1. The upper network accepts the first four letters, *happ*, and copies them to the output. From state 5 you could accept a *y* and have a complete word, or you could jump to state 6 to consider affixes. (The dashed link, indicating a jump, is not formally necessary but is useful for showing the break between the processing of the root form and the processing of the suffix.) For the word *happier,* you must jump to state 6. The next letter must be an *i,* which is transformed into a *y*. This is followed by a transition that uses no input (the empty symbol ε) and outputs a plus sign. From state 8, the input must be an *e,* and the output is also *e*. This must be followed by an *r* to get to state 10, which is encoded as a double circle indicating a possible end of word (that is, a success state for the FST). Thus this FST accepts the appropriate forms and outputs the desired sequence of morphemes.

The entire lexicon can be encoded as an FST that encodes all the legal input words and transforms them into morphemic sequences. The FSTs for the different suffixes need only be defined once, and all root forms that allow that

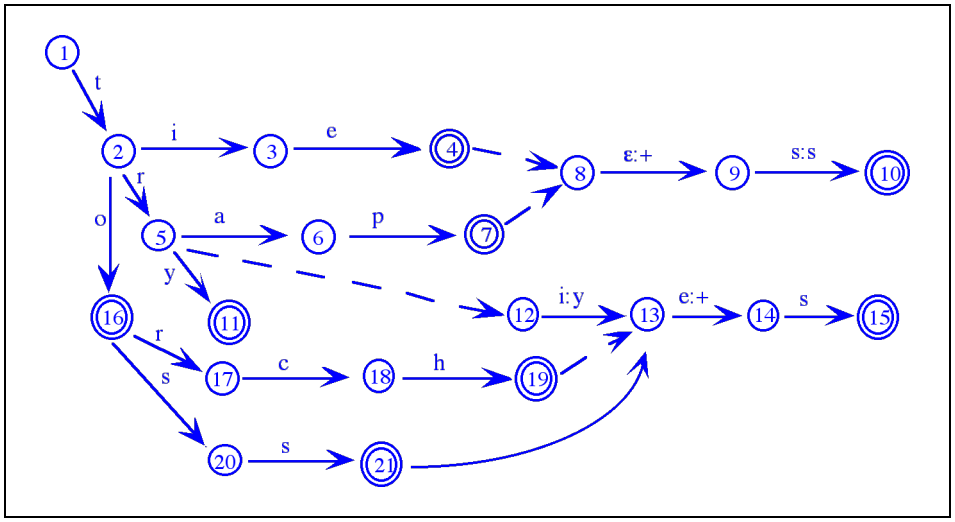


Figure 3.28 A fragment of an FST defining some nouns (singular and plural)

suffix can point to the same node. Words that share a common prefix (such as *torch, toss,* and *to*) also can share the same nodes, greatly reducing the size of the network. The FST in Figure 3.28 accepts the following words, which all start with *t*: *tie* (state 4), *ties* (10), *trap* (7), *traps* (10), *try* (11), *tries* (15), *to* (16), *torch* (19), *torches* (15), *toss* (21), and *tosses* (15). In addition, it outputs the appropriate sequence of morphemes.

Note that you may pass through acceptable states along the way when processing a word. For instance, with the input *toss* you would pass through state 15, indicating that *to* is a word. This analysis is not useful, however, because if *to* was accepted then the letters *ss* would not be accounted for.

Using such an FST, an input sentence can be processed into a sequence of morphemes. Occasionally, a word will be ambiguous and have multiple different decompositions into morphemes. This is rare enough, however, that we will ignore this minor complication throughout the book.

◦ 3.8 Grammars and Logic Programming

Another popular method of building a parser for CFGs is to encode the rules of the grammar directly in a logic programming language such as PROLOG. It turns out that the standard PROLOG interpretation algorithm uses exactly the same search strategy as the depth-first top-down parsing algorithm, so all that is needed is a way to reformulate context-free grammar rules as clauses in PROLOG. Consider the following CFG rule:

S → NP VP

This rule can be reformulated as an axiom that says, "A sequence of words is a legal S if it begins with a legal NP that is followed by a legal VP." If you number each word in a sentence by its position, you can restate this rule as: "There is an S between position p1 and p3, if there is a position p2 such that there is an NP between p1 and p2 and a VP between p2 and p3." In PROLOG this would be the following axiom, where variables are indicated as atoms with an initial capitalized letter:

```
s(P1, P3) :– np(P1, P2), vp(P2, P3)
```

To set up the process, add axioms listing the words in the sentence by their position. For example, the sentence *John ate the cat* is described by

```
word(john, 1, 2)
word(ate, 2, 3)
word(the, 3, 4)
word(cat, 4, 5)
```

The lexicon is defined by a set of predicates such as the following:

```
isart(the)
isname(john)
isverb(ate)
isnoun(cat)
```

Ambiguous words would produce multiple assertions—one for each syntactic category to which they belong.

For each syntactic category, you can define a predicate that is true only if the word between the two specified positions is of that category, as follows:

```
n(I, O) :– word(Word, I, O), isnoun(Word)
art(I, O) :– word(Word, I, O), isart(Word)
v(I, O) :– word(Word, I, O), isverb(Word)
name(I, O) :– word(Word, I, O), isname(Word)
```

Using the axioms in Figure 3.29, you can prove that *John ate the cat* is a legal sentence by proving `s(1, 5)`, as in Figure 3.30. In Figure 3.30, when there is a possibility of confusing different variables that have the same name, a prime (´) is appended to the variable name to make it unique. This proof trace is in the same format as the trace for the top-down CFG parser, as follows. The state of the proof at any time is the list of subgoals yet to be proven. Since the word positions are included in the goal description, no separate position column need be traced. The backup states are also lists of subgoals, maintained automatically by a system like PROLOG to implement backtracking. A typical trace of a proof in such a system shows only the current state at any time.

Because the standard PROLOG search strategy is the same as the depth-first top-down paring strategy, a parser built from PROLOG will have the same computational complexity, C^n, that is, the number of steps can be exponential in the

```
1.  s(P1, P3)  :–  np(P1, P2), vp(P2, P3)
2.  np(P1, P3)  :–  art(P1, P2), n(P2, P3)
3.  np(P1, P3)  :–  name(P1, P3)
4.  pp(P1, P3)  :–  p(P1, P2), np(P2, P3)
5.  vp(P1, P2)  :–  v(P1, P2)
6.  vp(P1, P3)  :–  v(P1, P2), np(P2, P3)
7.  vp(P1, P3)  :–  v(P1, P2), pp(P2, P3)
```

Figure 3.29 A PROLOG-based representation of Grammar 3.4

Step	Current State	Backup States	Comments
1.	s(1, 5)		
2.	np(1, P2) vp(P2, 5)		
3.	art(1, P2´) n(P2´, P2) vp(P2, 5)	name(1, P2) vp(P2, 5)	fails as no ART at position 1
4.	name(1, P2) vp(P2, 5)		
5.	vp(2, 5)		name(1, 2) proven
6.	v(2, 5)	v(2, P2) np(P2, 5) v(2, P2) pp(P2, 5)	fails as no verb spans positions 2 to 5
7.	v(2, P2) np(P2, 5)	v(2, P2) pp(P2, 5)	
8.	np(3, 5)	v(2, P2) pp(P2, 5)	v(2, 3) proven
9.	art(3, P2) n(P2, 5)	name(3, 5) v(2, P2) pp(P2, 5)	
10.	n(4, 5)	name(3, 5) v(2, P2) pp(P2, 5)	art(3, 4) proven
11.	√ proof succeeds	name(3, 5) v(2, P2) pp(P2, 5)	n(4, 5) proven

Figure 3.30 A trace of a PROLOG-based parse of *John ate the cat*

length of the input. Even with this worst-case analysis, PROLOG-based grammars can be quite efficient in practice. It is also possible to insert chart-like mechanisms to improve the efficiency of a grammar, although then the simple correspondence between context-free rules and PROLOG rules is lost. Some of these issues will be discussed in the next chapter.

It is worthwhile to try some simple grammars written in PROLOG to better understand top-down, depth-first search. By turning on the tracing facility, you can obtain a trace similar in content to that shown in Figure 3.30.

Summary

The two basic grammatical formalisms are context-free grammars (CFGs) and recursive transition networks (RTNs). A variety of parsing algorithms can be used for each. For instance, a simple top-down backtracking algorithm can be used for both formalisms and, in fact, the same algorithm can be used in the standard logic-programming-based grammars as well. The most efficient parsers use a chart-like structure to record every constituent built during a parse. By reusing this information later in the search, considerable work can be saved.

Related Work and Further Readings

There is a vast literature on syntactic formalisms and parsing algorithms. The notion of context-free grammars was introduced by Chomsky (1956) and has been studied extensively since in linguistics and in computer science. Some of this work will be discussed in detail later, as it is more relevant to the material in the following chapters.

Most of the parsing algorithms were developed in the mid-1960s in computer science, usually with the goal of analyzing programming languages rather than natural language. A classic reference for work in this area is Aho, Sethi, and Ullman (1986), or Aho and Ullman (1972), if the former is not available. The notion of a chart is described in Kay (1973; 1980) and has been adapted by many parsing systems since. The bottom-up chart parser described in this chapter is similar to the left-corner parsing algorithm in Aho and Ullman (1972), while the top-down chart parser is similar to that described by Earley (1970) and hence called the Earley algorithm.

Transition network grammars and parsers are described in Woods (1970; 1973) and parsers based on logic programming are described and compared with transition network systems in Pereira and Warren (1980). Winograd (1983) discusses most of the approaches described here from a slightly different perspective, which could be useful if a specific technique is difficult to understand. Gazdar and Mellish (1989a; 1989b) give detailed descriptions of implementations of parsers in LISP and in PROLOG. In addition, descriptions of transition network parsers can be found in many introductory AI texts, such as Rich and Knight (1992), Winston (1992), and Charniak and McDermott (1985). These books also contain descriptions of the search techniques underlying many of the parsing algorithms. Norvig (1992) is an excellent source on AI programming techniques.

The best sources for work on computational morphology are two books: Sproat (1992) and Ritchie et al. (1992). Much of the recent work on finite state models has been based on the KIMMO system (Koskenniemi, 1983). Rather than requiring the construction of a huge network, KIMMO uses a set of FSTs which are run in parallel; that is, all of them must simultaneously accept the input and agree on the output. Typically, these FSTs are expressed using an abstract

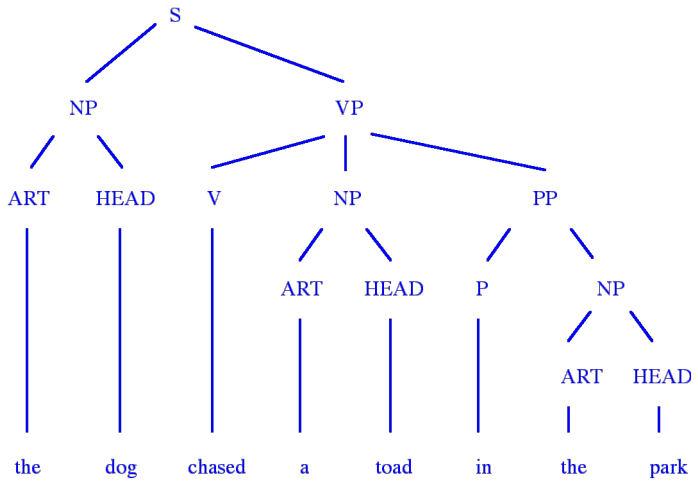
language that allows general morphological rules to be expressed concisely. A compiler can then be used to generate the appropriate networks for the system.

Finite state models are useful for a wide range of processing tasks besides morphological analysis. Blank (1989), for instance, is developing a grammar for English using only finite state methods. Finite state grammars are also used extensively in speech recognition systems.

Exercises for Chapter 3

1. (*easy*)

a. Express the following tree in the list notation in Section 3.1.



b. Is there a tree structure that could not be expressed as a list structure? How about a list structure that could not be expressed as a tree?

2. (*easy*) Given the CFG in Grammar 3.4, define an appropriate lexicon and show a trace in the format of Figure 3.5 of a top-down CFG parse of the sentence *The man walked the old dog*.

3. (*easy*) Given the RTN in Grammar 3.19 and a lexicon in which *green* can be an adjective or a noun, show a trace in the format of Figure 3.21 of a top-down RTN parse of the sentence *The green faded*.

4. (*easy*) Given the PROLOG-based grammar defined in Figure 3.29, show a trace in the format of Figure 3.30 of the proof that the following is a legal sentence: *The cat ate John*.

5. (*medium*) Map the following context-free grammar into an equivalent recursive transition network that uses only three networks—an S, NP, and PP network. Make your networks as small as possible.

S → NP VP
VP → V
VP → V NP
VP → V PP
NP → ART NP2
NP → NP2
NP2 → N
NP2 → ADJ NP2
NP2 → NP3 PREPS
NP3 → N
PREPS → PP
PREPS → PP PREPS
PP → NP

6. (*medium*) Given the CFG in Exercise 5 and the following lexicon, construct a trace of a pure top-down parse and a pure bottom-up parse of the sentence *The herons fly in groups*. Make your traces as clear as possible, select the rules in the order given in Exercise 5, and indicate all parts of the search. The lexicon entries for each word are

the: ART
herons: N
fly: N V ADJ
in: P
groups: N V

7. (*medium*) Consider the following grammar:

S → ADJS N
S → N
ADJS → ADJS ADJ
ADJS → ADJ

Lexicon: ADJ: red, N: house

a. What happens to the top-down depth-first parser operating on this grammar trying to parse the input *red red*? In particular, state whether the parser succeeds, fails, or never stops.

b. How about a top-down breadth-first parser operating on the same input *red red*?

c. How about a top-down breadth-first parser operating on the input *red house*?

d. How about a bottom-up depth-first parser on *red house*?

e. For the cases where the parser fails to stop, give a grammar that is equivalent to the one shown in this exercise and that is parsed correctly. (Correct behavior includes failing on unacceptable phrases as well as succeeding on acceptable ones.)

f. With the new grammar in part (e), do all the preceding parsers now operate correctly on the two phrases *red red* and *red house*?

8. (*medium*) Consider the following CFG:

S → NP V
S → NP AUX V
NP → ART N

Trace one of the chart parsers in processing the sentence

$_1$ The $_2$ man $_3$ is $_4$ laughing $_5$

with the lexicon entries:

the: ART
man: N
is: AUX
laughing: V

Show every step of the parse, giving the parse stack, and drawing the chart each time a nonterminal constituent is added to the chart.

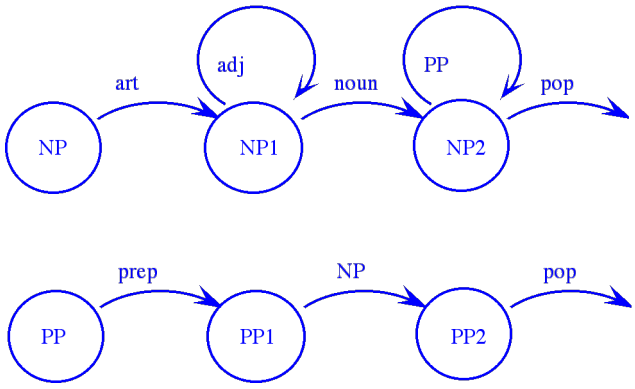
9. (*medium*) Consider the following CFG that generates sequences of letters:

s → a x c
s → b x c
s → b x d
s → b x e
s → c x e
x → f x
x → g

a. If you had to write a parser for this grammar, would it be better to use a pure top-down or a pure bottom-up approach? Why?

b. Trace the parser of your choice operating on the input *bffge*.

10. (*medium*) Consider the following CFG and RTN:



NP → ART NP1
NP1 → ADJ N PPS
PPS → PP
PPS → PP PPS
PP → P NP

a. State two ways in which the languages described by these two grammars differ. For each, give a sample sentence that is recognized by one grammar but not the other and that demonstrates the difference.

b. Write a new CFG equivalent to the RTN shown here.

c. Write a new RTN equivalent to the CFG shown here.

11. (*hard*) Consider the following sentences:

List A		**List B**	
i.	Joe is reading the book.	*i*.	*Joe has reading the book.
ii.	Joe had won a letter.	*ii*.	*Joe had win.
iii.	Joe has to win.	*iii*.	*Joe winning.
iv.	Joe will have the letter.	*iv*.	*Joe will had the letter.
v.	The letter in the book was read.	*v*.	*The book was win by Joe.
vi.	The letter must have been in the book by Joe.	*vi*.	*Joe will can be mad.
vii.	The man could have had one.	*vii*.	*The man can have having one.

a. Write a context-free grammar that accepts all the sentences in list A while rejecting the sentences in list B. You may find it useful to make reference to the grammatical forms of verbs discussed in Chapter 2.

b. Implement one of the chart-based parsing strategies and, using the grammar specified in part (a), demonstrate that your parser correctly accepts all the sentences in A and rejects those in B. You should maintain enough information in each entry on the chart so that you can reconstruct the parse tree for each possible interpretation. Make sure your method of recording the structure is well documented and clearly demonstrated.

c. List three (distinct) grammatical forms that would not be recognized by a parser implementing the grammar in part (a). Provide an example of your own for each of these grammatical forms.