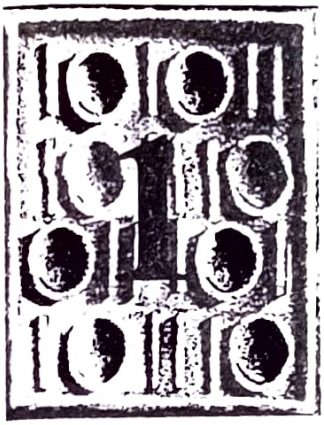
CHAPTER



Introduction to Natural Language Understanding

This chapter describes the field of natural language understanding and introduces some basic distinctions. Section 1.1 discusses how natural language understanding research fits into the study of language in general. Section 1.2 discusses some applications of natural language understanding systems and considers what it means for a system to understand language. Section 1.3 describes how you might evaluate whether a system understands language. Section 1.4 introduces a few basic distinctions that are made when studying language, and Section 1.5 discusses how computational systems often realize these distinctions. Finally, Section 1.6 discusses how natural language systems are generally organized, and introduces the particular organization assumed throughout this book.

1.1 The Study of Language

Language is one of the fundamental aspects of human behavior and is a crucial component of our lives. In written form it serves as a long-term record of knowledge from one generation to the next. In spoken form it serves as our primary means of coordinating our day-to-day behavior with others. This book describes research about how language comprehension and production work. The goal of this research is to create computational models of language in enough detail that you could write computer programs to perform various tasks involving natural language. The ultimate goal is to be able to specify models that approach human performance in the linguistic tasks of reading, writing, hearing, and speaking. This book, however, is not concerned with problems related to the specific medium used, whether handwriting, keyboard input, or speech. Rather, it is concerned with the processes of comprehending and using language once the words are recognized. Computational models are useful both for scientific purposes—for exploring the nature of linguistic communication—and for practical purposes—for enabling effective human-machine communication.

Language is studied in several different academic disciplines. Each discipline defines its own set of problems and has its own methods for addressing them. The linguist, for instance, studies the structure of language itself, considering questions such as why certain combinations of words form sentences but others do not, and why a sentence can have some meanings but not others. The psycholinguist, on the other hand, studies the processes of human language production and comprehension, considering questions such as how people identify the appropriate structure of a sentence and when they decide on the appropriate meaning for words. The philosopher considers how words can mean anything at all and how they identify objects in the world. Philosophers also consider what it means to have beliefs, goals, and intentions, and how these cognitive capabilities relate to language. The goal of the computational linguist is to develop a computational theory of language, using the notions of algorithms and data structures from computer science. Of course, to build a computational model, you must take advantage of what is known from all the other disciplines. Figure 1.1 summarizes these different approaches to studying language.

Discipline	Typical Problems	Tools
Linguists	How do words form phrases and sentences? What constrains the possible meanings for a sentence?	Intuitions about well-formedness and meaning; mathematical models of structure (for example, formal language theory, model theoretic semantics)
Psycholinguists	How do people identify the structure of sentences? How are word meanings identified? When does understanding take place?	Experimental techniques based on measuring human performance; statistical analysis of observations
Philosophers	What is meaning, and how do words and sentences acquire it? How do words identify objects in the world?	Natural language argumentation using intuition about counter-examples; mathematical models (for example, logic and model theory)
Computational Linguists	How is the structure of sentences identified? How can knowledge and reasoning be modeled? How can language be used to accomplish specific tasks?	Algorithms, data structures; formal models of representation and reasoning; AI techniques (search and representation methods)

Figure 1.1 The major disciplines studying language

As previously mentioned, there are two motivations for developing computational models. The scientific motivation is to obtain a better understanding of how language works. It recognizes that any one of the other traditional disciplines does not have the tools to completely address the problem of how language comprehension and production work. Even if you combine all the approaches, a comprehensive theory would be too complex to be studied using traditional methods. But we may be able to realize such complex theories as computer programs and then test them by observing how well they perform. By seeing where they fail, we can incrementally improve them. Computational models may provide very specific predictions about human behavior that can then be explored by the psycholinguist. By continuing in this process, we may eventually acquire a deep understanding of how human language processing occurs. To realize such a dream will take the combined efforts of linguists, psycholinguists, philosophers, and computer scientists. This common goal has motivated a new area of interdisciplinary research often called cognitive science.

The practical, or technological, motivation is that natural language processing capabilities would revolutionize the way computers are used. Since most of human knowledge is recorded in linguistic form, computers that could understand natural language could access all this information. In addition, natural language interfaces to computers would allow complex systems to be accessible to

BOX 1.1 Boxes and Optional Sections

This book uses several techniques to allow you to identify what material is central and what is optional. In addition, optional material is sometimes classified as advanced, indicating that you may need additional background not covered in this book to fully appreciate the text. Boxes, like this one, always contain optional material, either providing more detail on a particular approach discussed in the main text or discussing additional issues that are related to the text. Sections and subsections may be marked as optional by means of an open dot (∘) before the heading. Optional sections provide more breadth and depth to chapters, but are not necessary for understanding material in later chapters. Depending on your interests and focus, you can choose among the optional sections to fill out the core material presented in the regular sections. In addition, there are dependencies between the chapters, so that entire chapters can be skipped if the material does not address your interests. The chapter dependencies are not marked explicitly in the text, but a chart of dependencies is given in the preface.

everyone. Such systems would be considerably more flexible and intelligent than is possible with current computer technology. For technological purposes it does not matter if the model used reflects the way humans process language. It only matters that it works.

This book takes a middle ground between the scientific and technological goals. On the one hand, this reflects a belief that natural language is so complex that an *ad hoc* approach without a well-specified underlying theory will not be successful. Thus the technological goal cannot be realized without using sophisticated underlying theories on the level of those being developed by linguists, psycholinguists, and philosophers. On the other hand, the present state of knowledge about natural language processing is so preliminary that attempting to build a cognitively correct model is not feasible. Rather, we are still attempting to construct any model that appears to work.

The goal of this book is to describe work that aims to produce linguistically motivated computational models of language understanding and production that can be shown to perform well in specific example domains. While the book focuses on computational aspects of language processing, considerable space is spent introducing the relevant background knowledge from the other disciplines that motivates and justifies the computational approaches taken. It assumes only a basic knowledge of programming, although the student with some background in linguistics, artificial intelligence (AI), and logic will appreciate additional subtleties in the development.

1.2 Applications of Natural Language Understanding

A good way to define natural language research is to consider the different applications that researchers work on. As you consider these examples, it will

also be a good opportunity to consider what it would mean to say that a computer system understands natural language. The applications can be divided into two major classes: text-based applications and dialogue-based applications.

Text-based applications involve the processing of written text, such as books, newspapers, reports, manuals, e-mail messages, and so on. These are all reading-based tasks. Text-based natural language research is ongoing in applications such as

- finding appropriate documents on certain topics from a database of texts (for example, finding relevant books in a library)
- extracting information from messages or articles on certain topics (for example, building a database of all stock transactions described in the news on a given day)
- translating documents from one language to another (for example, producing automobile repair manuals in many different languages)
- summarizing texts for certain purposes (for example, producing a 3-page summary of a 1000-page government report)

Not all systems that perform such tasks must be using natural language understanding techniques in the way we mean in this book. For example, consider the task of finding newspaper articles on a certain topic in a large database. Many techniques have been developed that classify documents by the presence of certain keywords in the text. You can then retrieve articles on a certain topic by looking for articles that contain the keywords associated with that topic. Articles on law, for instance, might contain the words *lawyer, court, sue, affidavit*, and so on, while articles on stock transactions might contain words such as *stocks, takeover, leveraged buyout, options,* and so on. Such a system could retrieve articles on any topic that has been predefined by a set of keywords. Clearly, we would not say that this system is understanding the text; rather, it is using a simple matching technique. While such techniques may produce useful applications, they are inherently limited. It is very unlikely, for example, that they could be extended to handle complex retrieval tasks that are easily expressed in natural language, such as the query *Find me all articles on leveraged buyouts involving more than 100 million dollars that were attempted but failed during 1986 and 1990.* To handle such queries, the system would have to be able to extract enough information from each article in the database to determine whether the article meets the criteria defined by the query; that is, it would have to build a representation of the information in the articles and then use the representation to do the retrievals. This identifies a crucial characteristic of an understanding system: it must compute some representation of the information that can be used for later inference.

Consider another example. Some machine translation systems have been built that are based on pattern matching; that is, a sequence of words in one language is associated with a sequence of words in another language. The

translation is accomplished by finding the best set of patterns that match the input and producing the associated output in the other language. This technique can produce reasonable results in some cases but sometimes produces completely wrong translations because of its inability to use an understanding of content to disambiguate word senses and sentence meanings appropriately. In contrast, other machine translation systems operate by producing a representation of the meaning of each sentence in one language, and then producing a sentence in the other language that realizes the same meaning. This latter approach, because it involves the computation of a representation of meaning, is using natural language understanding techniques.

One very attractive domain for text-based research is story understanding. In this task the system processes a story and then must answer questions about it. This is similar to the type of reading comprehension tests used in schools and provides a very rich method for evaluating the depth of understanding the system is able to achieve.

Dialogue-based applications involve human-machine communication. Most naturally this involves spoken language, but it also includes interaction using keyboards. Typical potential applications include

- question-answering systems, where natural language is used to query a database (for example, a query system to a personnel database)
- automated customer service over the telephone (for example, to perform banking transactions or order items from a catalogue)
- tutoring systems, where the machine interacts with a student (for example, an automated mathematics tutoring system)
- spoken language control of a machine (for example, voice control of a VCR or computer)
- general cooperative problem-solving systems (for example, a system that helps a person plan and schedule freight shipments)

Some of the problems faced by dialogue systems are quite different than in text-based systems. First, the language used is very different, and the system needs to participate actively in order to maintain a natural, smooth-flowing dialogue. Dialogue requires the use of acknowledgments to verify that things are understood, and an ability to both recognize and generate clarification sub-dialogues when something is not clearly understood. Even with these differences, however, the basic processing techniques are fundamentally the same.

It is important to distinguish the problems of speech recognition from the problems of language understanding. A speech recognition system need not involve any language understanding. For instance, voice-controlled computers and VCRs are entering the market now. These do not involve natural language understanding in any general way. Rather, the words recognized are used as commands, much like the commands you send to a VCR using a remote control. Speech recognition is concerned only with identifying the words spoken from a

given speech signal, not with understanding how words are used to communicate. To be an understanding system, the speech recognizer would need to feed its input to a natural language understanding system, producing what is often called a spoken language understanding system.

With few exceptions, all the techniques discussed in this book are equally relevant for text-based and dialogue-based language understanding, and apply equally well whether the input is text, keyboard, or speech. The key characteristic of any understanding system is that it represents the meaning of sentences in some representation language that can be used later for further processing.

1.3 Evaluating Language Understanding Systems

As you can see, what counts as understanding might vary from application to application. If this is so, how can you tell if a system works? One obvious way to evaluate a system is to run the program and see how well it performs the task it was designed to do. If the program is meant to answer questions about a database of facts, you might ask it questions to see how good it is at producing the correct answers. If the system is designed to participate in simple conversations on a certain topic, you might try conversing with it. This is called **black box evaluation** because it evaluates system performance without looking inside to see how it works. While ultimately this method of evaluation may be the best test of a system's capabilities, it is problematic in the early stages of research because early evaluation results can be misleading. Sometimes the techniques that produce the best results in the short term will not lead to the best results in the long term. For instance, if the overall performance of all known systems in a given application is uniformly low, few conclusions can be drawn. The fact that one system was correct 50 percent of the time while another was correct only 40 percent of the time says nothing about the long-term viability of either approach. Only when the success rates become high, making a practical application feasible, can much significance be given to overall system performance measures.

An alternative method of evaluation is to identify various subcomponents of a system and then evaluate each one with appropriate tests. This is called **glass box evaluation** because you look inside at the structure of the system. The problem with glass box evaluation is that it requires some consensus on what the various components of a natural language system should be. Achieving such a consensus is an area of considerable activity at the present.

Despite the difficulties, evaluations are of crucial importance to the field, as very little can be concluded from examining a few examples that a particular system can handle. If you cannot interact freely with the program yourself but must depend on transcripts of others' interactions, you can easily be mislead into overestimating a program's abilities. It is very hard for people not to attribute much more intelligence to the program than is actually present.

Consider a concrete example—the ELIZA program that was developed in the mid-1960s at MIT. This program must be one of the most popular AI

User:	ELIZA:
Men are all alike.	In what way?
They're always bugging us about something or other.	Can you think of a specific example?
Well, my boyfriend made me come here.	Your boyfriend made you come here.
He says I'm depressed much of the time.	I am sorry to hear you are depressed.
It's true, I am unhappy.	Do you think coming here will help you not to be unhappy?
I need some help; that much seems certain.	What would it mean to you if you got some help?
Perhaps I could learn to get along with my mother.	Tell me more about your family.
My mother takes care of me.	Who else in your family takes care of you?
My father.	Your father.
You are like my father in some ways.	What resemblance do you see?

Figure 1.2 A dialogue with ELIZA

programs of all time, and versions of it exist for most machines, including most personal computers. ELIZA was never claimed to embody a theory of language comprehension and production, but it serves as an excellent example as its behavior initially seems impressive. The system plays the role of a therapist and, to obtain the best results, the user should correspondingly play the role of a patient. Figure 1.2 presents a transcript of this system in operation. Given this transcript, or even playing with the system yourself for a few minutes, ELIZA's performance certainly seems impressive.

Here is a simple description of how ELIZA works. There is a database of particular words that are called **keywords**. For each keyword, the system stores an integer, a pattern to match against the input, and a specification of the output. The algorithm is as follows: Given a sentence S, find a keyword in S whose pattern matches S. If there is more than one keyword, pick the one with the highest integer value. Use the output specification that is associated with this keyword to generate the next sentence. If there are no keywords, generate an innocuous continuation statement, such as *Tell me more* or *Go on.*

Figure 1.3 shows a fragment of a database of keywords. In this database a pattern consists of words and variables. The prefix ? before a letter indicates a variable, which can match any sequence of words. For example, the pattern

?X are you ?Y

would match the sentence *Why are you looking at me?*, where the variable ?X matches *Why* and ?Y matches *looking at me.* The output specification may also use the same variables. In this case, ELIZA inserts the words that match the variables in the input into the output after making some minor changes in the

Word	Rank	Pattern	Outputs
alike	10	?X	In what way? What resemblance do you see?
are	3	?X are you ?Y	Would you prefer it if I weren't ?Y?
	3	?X are ?Y	What if they were not ?Y?
always	5	?X	Can you think of a specific example? When? Really, always?
what	2	?X	Why do you ask? Does that interest you?

Figure 1.3 Sample data from ELIZA

pronouns (for example, replacing *me* with *you*). Thus, for the pattern above, if the output specification is

Would you prefer it if I weren't ?Y?

the rule would generate a response *Would you prefer it if I weren't looking at you?* When the database lists multiple output specifications for a given pattern, ELIZA selects a different one each time a keyword rule is used, thereby preventing unnatural repetition in the conversation. Using these rules, you can see how ELIZA produced the first two exchanges in the conversation in Figure 1.2. ELIZA generated the first response from the first output of the keyword *alike* and the second response from the first output of the keyword *always.*

This description covers all of the essential points of the program. You will probably agree that the program does not understand the conversation it is participating in. Rather, it is a collection of tricks. Why then does ELIZA appear to function so well? There are several reasons. Perhaps the most important reason is that, when people hear or read a sequence of words that they understand as a sentence, they attribute meaning to the sentence and assume that the person (or machine) that produced the sentence actually intended that meaning. People are extremely good at distinguishing word meanings and interpreting sentences to fit the context. Thus ELIZA appears to be intelligent because you use your own intelligence to make sense of what it says.

Other crucial characteristics of the conversational setting also aid in sustaining the illusion of intelligence. For instance, the system does not need any world knowledge because it never has to make a claim, support an argument, or answer a question. Rather, it simply asks a series of questions. Except in a patient-therapist situation, this would be unacceptable. ELIZA evades all direct questions by responding with another question, such as *Why do you ask?* There is no way to force the program to say something concrete about any topic.

Even in such a restricted situation, however, it is relatively easy to demonstrate that the program does not understand. It sometimes produces completely off-the-wall responses. For instance, if you say *Necessity is the mother of invention,* it might respond with *Tell me more about your family,* based on its pattern for the word *mother.* In addition, since ELIZA has no knowledge about the structure of language, it accepts gibberish just as readily as valid sentences. If you enter *Green the adzabak are the a ran four,* ELIZA will respond with something like *What if they were not the a ran four?* Also, as a conversation progresses, it becomes obvious that the program does not retain any of the content in the conversation. It begins to ask questions that are inappropriate in light of earlier exchanges, and its responses in general begin to show a lack of focus. Of course, if you are not able to play with the program and must depend only on transcripts of conversations by others, you would have no way of detecting these flaws, unless they are explicitly mentioned.

Suppose you need to build a natural language program for a certain application in only six months. If you start to construct a general model of language understanding, it will not be completed in that time frame and so will perform miserably on the tests. An ELIZA-like system, however, could easily produce behavior like that previously discussed with less than a few months of programming and will appear to far outperform the other system in testing. The differences will be especially marked if the test data only includes typical domain interactions that are not designed to test the limits of the system. Thus, if we take short-term performance as our only criteria of progress, everyone will build and fine-tune ELIZA-style systems, and the field will not progress past the limitations of the simple approach.

To avoid this problem, either we have to accept certain theoretical assumptions about the architecture of natural language systems and develop specific evaluation measures for different components, or we have to discount overall evaluation results until some reasonably high level of performance is obtained. Only then will cross-system comparisons be
term success in the field.

1.4 The Different Levels of Language A

A natural language system must use considei
of the language itself, including what the words are, how words combine to form sentences, what the words mean, how word meanings contribute to sentence meanings, and so on. However, we cannot completely account for linguistic behavior without also taking into account another aspect of what makes humans intelligent—their general world knowledge and their reasoning abilities. For example, to answer questions or to participate in a conversation, a person not only must know a lot about the structure of the language being used, but also must know about the world in general and the conversational setting in particular.

senses of a word when understanding a sentence, a program must explicitly consider them one by one.

To represent meaning, we must have a more precise language. The tools to do this come from mathematics and logic and involve the use of formally specified representation languages. Formal languages are specified from very simple building blocks. The most fundamental is the notion of an atomic symbol, which is distinguishable from any other atomic symbol simply based on how it is written. Useful representation languages have the following two properties:

- The representation must be precise and unambiguous. You should be able to express every distinct reading of a sentence as a distinct formula in the representation.
- The representation should capture the intuitive structure of the natural language sentences that it represents. For example, sentences that appear to be structurally similar should have similar structural representations, and the meanings of two sentences that are paraphrases of each other should be closely related to each other.

Several different representations will be used that correspond to some of the levels of analysis discussed in the last section. In particular, we will develop formal languages for expressing syntactic structure, for context-independent word and sentence meanings, and for expressing general world knowledge.

Syntax: Representing Sentence Structure

The syntactic structure of a sentence indicates the way that words in the sentence are related to each other. This structure indicates how the words are grouped together into phrases, what words modify what other words, and what words are of central importance in the sentence. In addition, this structure may identify the types of relationships that exist between phrases and can store other information about the particular sentence structure that may be needed for later processing. For example, consider the following sentences:

1. John sold the book to Mary.
2. The book was sold to Mary by John.

These sentences share certain structural properties. In each, the noun phrases are *John, Mary,* and *the book,* and the act described is some selling action. In other respects, these sentences are significantly different. For instance, even though both sentences are always either true or false in the exact same situations, you could only give sentence 1 as an answer to the question *What did John do for Mary?* Sentence 2 is a much better continuation of a sentence beginning with the phrase *After it fell in the river,* as sentences 3 and 4 show. Following the standard convention in linguistics, this book will use an asterisk (*) before any example of an ill-formed or questionable sentence.

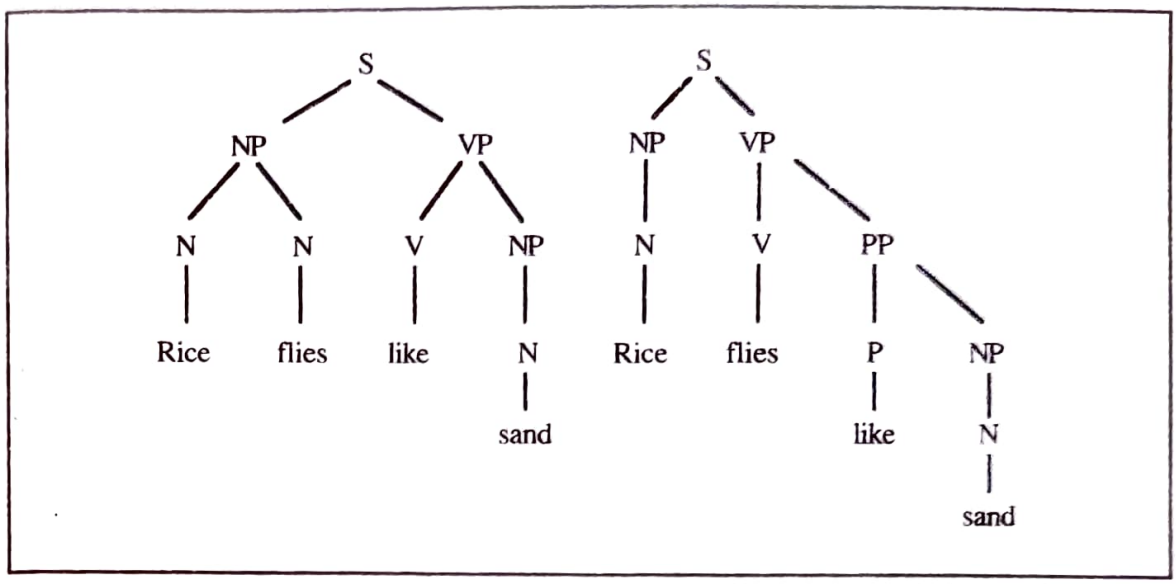


Figure 1.4 Two structural representations of *Rice flies like sand.*

3. *After it fell in the river, John sold Mary the book.
4. After it fell in the river, the book was sold to Mary by John.

Many other structural properties can be revealed by considering sentences that are not well-formed. Sentence 5 is ill-formed because the subject and the verb do not agree in number (the subject is singular and the verb is plural), while 6 is ill-formed because the verb *put* requires some modifier that describes where John put the object.

5. *John are in the corner.
6. *John put the book.

Making judgments on grammaticality is not a goal in natural language understanding. In fact, a robust system should be able to understand ill-formed sentences whenever possible. This might suggest that agreement checks can be ignored, but this is not so. Agreement checks are essential for eliminating potential ambiguities. Consider sentences 7 and 8, which are identical except for the number feature of the main verb, yet represent two quite distinct interpretations.

7. Flying planes are dangerous.
8. Flying planes is dangerous.

If you did not check subject-verb agreement, these two sentences would be indistinguishable and ambiguous. You could find similar examples for every syntactic feature that this book introduces and uses.

Most syntactic representations of language are based on the notion of context-free grammars, which represent sentence structure in terms of what phrases are subparts of other phrases. This information is often presented in a tree form, such as the one shown in Figure 1.4, which shows two different structures

for the sentence *Rice flies like sand.* In the first reading, the sentence is formed from a noun phrase (NP) describing a type of fly, rice flies, and a verb phrase (VP) that asserts that these flies like sand. In the second structure, the sentence is formed from a noun phrase describing a type of substance, rice, and a verb phrase stating that this substance flies like sand (say, if you throw it). The two structures also give further details on the structure of the noun phrase and verb phrase and identify the part of speech for each word. In particular, the word *like* is a verb (V) in the first reading and a preposition (P) in the second.

The Logical Form

The structure of a sentence doesn't reflect its meaning, however. For example, the NP *the catch* can have different meanings depending on whether the speaker is talking about a baseball game or a fishing expedition. Both these interpretations have the same syntactic structure, and the different meanings arise from an ambiguity concerning the sense of the word *catch*. Once the correct sense is identified, say the fishing sense, there still is a problem in determining what fish are being referred to. The intended meaning of a sentence depends on the situation in which the sentence is produced. Rather than combining all these problems, this book will consider each one separately. The division is between context-independent meaning and context-dependent meaning. The fact that *catch* may refer to a baseball move or the results of a fishing expedition is knowledge about English and is independent of the situation in which the word is used. On the other hand, the fact that a particular noun phrase *the catch* refers to what Jack caught when fishing yesterday is contextually dependent. The representation of the context-independent meaning of a sentence is called its **logical form.**

The logical form encodes possible word senses and identifies the semantic relationships between the words and phrases. Many of these relationships are often captured using an abstract set of semantic relationships between the verb and its NPs. In particular, in both sentences 1 and 2 previously given, the action described is a selling event, where *John* is the seller, *the book* is the object being sold, and *Mary* is the buyer. These roles are instances of the abstract semantic roles AGENT, THEME, and TO-POSS (for final possessor), respectively.

Once the semantic relationships are determined, some word senses may be impossible and thus eliminated from consideration. Consider the sentence

9. Jack invited Mary to the Halloween ball.

The word *ball,* which by itself is ambiguous between the plaything that bounces and the formal dance event, can only take the latter sense in sentence 9, because the verb *invite* only makes sense with this interpretation. One of the key tasks in semantic interpretation is to consider what combinations of the individual word meanings can combine to create coherent sentence meanings. Exploiting such

interconnections between word meanings can greatly reduce the number of possible word senses for each word in a given sentence.

The Final Meaning Representation

The final representation needed is a general knowledge representation (KR), which the system uses to represent and reason about its application domain. This is the language in which all the specific knowledge based on the application is represented. The goal of contextual interpretation is to take a representation of the structure of a sentence and its logical form, and to map this into some expression in the KR that allows the system to perform the appropriate task in the domain. In a question-answering application, a question might map to a database query, in a story-understanding application, a sentence might map into a set of expressions that represent the situation that the sentence describes.

For the most part, we will assume that the first-order predicate calculus (FOPC) is the final representation language because it is relatively well known, well studied, and is precisely defined. While some inadequacies of FOPC will be examined later, these inadequacies are not relevant for most of the issues to be discussed.

1.6 The Organization of Natural Language Understanding Systems

This book is organized around the three levels of representation just discussed: syntactic structure, logical form, and the final meaning representation. Separating the problems in this way will allow you to study each problem in depth without worrying about other complications. Actual systems are usually organized slightly differently, however. In particular, Figure 1.5 shows the organization that this book assumes.

As you can see, there are interpretation processes that map from one representation to the other. For instance, the process that maps a sentence to its syntactic structure and logical form is called the **parser**. It uses knowledge about word and word meanings (the **lexicon**) and a set of rules defining the legal structures (the **grammar**) in order to assign a syntactic structure and a logical form to an input sentence. An alternative organization could perform syntactic processing first and then perform semantic interpretation on the resulting structures. Combining the two, however, has considerable advantages because it leads to a reduction in the number of possible interpretations, since every proposed interpretation must simultaneously be syntactically and semantically well formed. For example, consider the following two sentences:

10. Visiting relatives can be trying.
11. Visiting museums can be trying.

These two sentences have identical syntactic structure, so both are syntactically ambiguous. In sentence 10, the subject might be relatives who are visiting you or

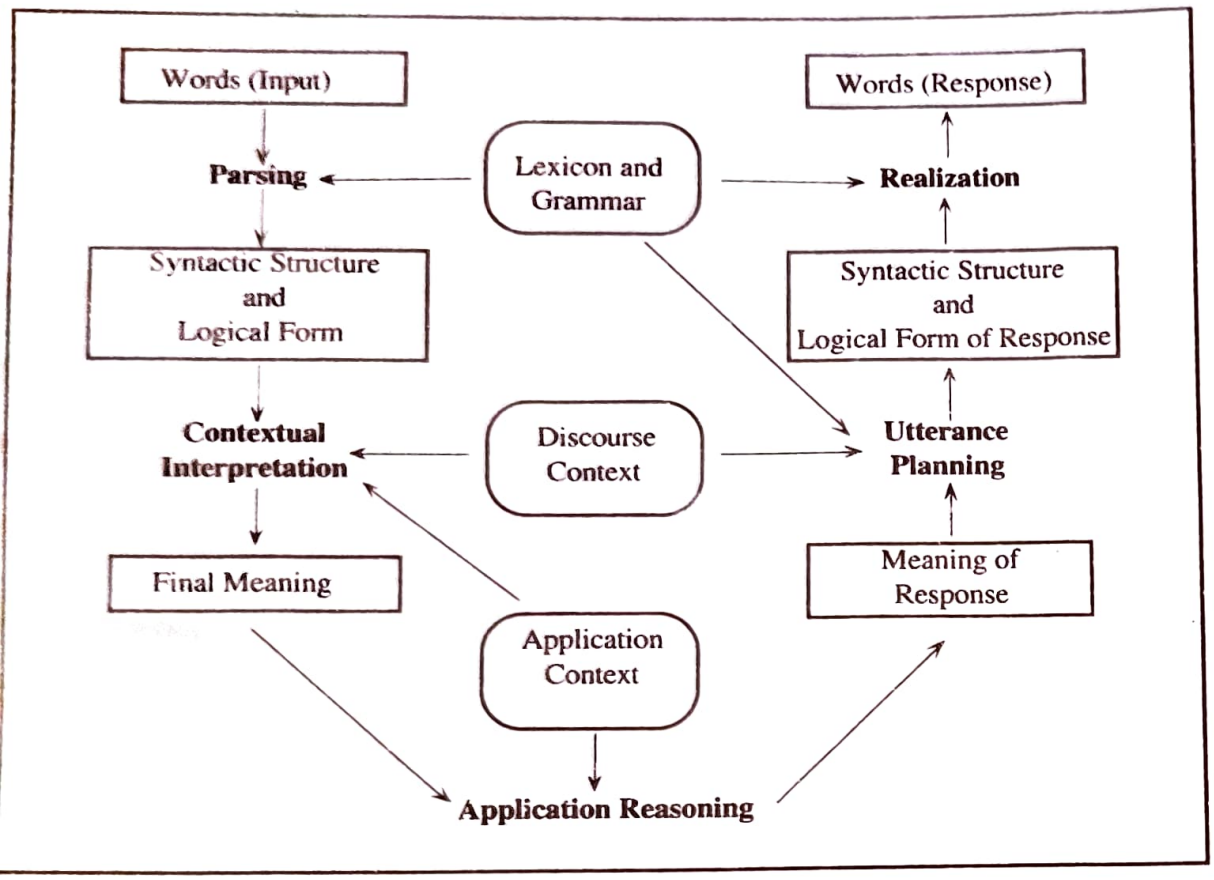


Figure 1.5 The flow of information

the event of you visiting relatives. Both of these alternatives are semantically valid, and you would need to determine the appropriate sense by using the contextual mechanism. However, sentence 11 has only one possible semantic interpretation, since museums are not objects that can visit other people; rather they must be visited. In a system with separate syntactic and semantic processing, there would be two syntactic interpretations of sentence 11, one of which the semantic interpreter would eliminate later. If syntactic and semantic processing are combined, however, the system will be able to detect the semantic anomaly as soon as it interprets the phrase *visiting museums*, and thus will never build the incorrect syntactic structure in the first place. While the savings here seem small, in a realistic application a reasonable sentence may have hundreds of possible syntactic structures, many of which are semantically anomalous.

Continuing through Figure 1.5, the process that transforms the syntactic structure and logical form into a final meaning representation is called contextual processing. This process includes issues such as identifying the objects referred to by noun phrases such as definite descriptions (for example, *the man*) and pronouns, the analysis of the temporal aspects of the new information conveyed by the sentence, the identification of the speaker's intention (for example, whether *Can you lift that rock* is a yes/no question or a request), as well as all the

inferential processing required to interpret the sentence appropriately within the application domain. It uses knowledge of the discourse context (determined by the sentences that preceded the current one) and knowledge of the application to produce a final representation.

The system would then perform whatever reasoning tasks are appropriate for the application. When this requires a response to the user, the meaning that must be expressed is passed to the generation component of the system. It uses knowledge of the discourse context, plus information on the grammar and lexicon, to plan the form of an utterance, which then is mapped into words by a realization process. Of course, if this were a spoken language application, the words would not be the final input and output, but rather would be the output of a speech recognizer and the input to a speech synthesizer, as appropriate.

While this text focuses primarily on language understanding, notice that the same levels of knowledge are also used for the generation task as well. For instance, knowledge of syntactic structure is encoded in the grammar. This grammar can be used either to identify the structure of a given sentence or to realize a structure as a sequence of words. A grammar that supports both processes is called a **bidirectional grammar**. While most researchers agree that bidirectional grammars are the preferred model, in actual practice grammars are often tailored for the understanding task or the generation task. This occurs because different issues are important for each task, and generally any given researcher focuses just on the problems related to their specific task. But even when the actual grammars differ between understanding and generation, the grammatical formalisms used remain the same.

Summary

This book describes computational theories of natural language understanding. The principal characteristic of understanding systems is that they compute representations of the meanings of sentences and use these representations in reasoning tasks. Three principal levels of representation were introduced that correspond to the three main subparts of this book. Syntactic processing is concerned with the structural properties of sentences; semantic processing computes a logical form that represents the context-independent meaning of the sentence; and contextual processing connects language to the application domain.

Related Work and Further Readings

A good idea of work in the field can be obtained by reading two articles in Shapiro (1992), under the headings "Computational Linguistics" and "Natural Language Understanding." There are also articles on specialized subareas such as machine translation, natural language interfaces, natural language generation, and so on. Longer surveys on certain areas are also available. Slocum (1985) gives a

survey of machine translation, and Perrault and Grosz (1986) give a survey of natural language interfaces.

You can find a description of the ELIZA program that includes the transcript of the dialogue in Figure 1.2 in Weizenbaum (1966). The basic technique of using template matching was developed further in the PARRY system, as described in the paper by Colby in Schank and Colby (1973). That same book also contains descriptions of early natural language systems, including those by Winograd and by Schank. Another important early system is the LUNAR system, an overview of which can be found in Woods (1977). For another perspective on the AI approach to natural language, refer to the introduction in Winograd (1983).

Exercises for Chapter 1

1. (*easy*) Define a set of data rules for ELIZA that would generate the first seven exchanges in the conversation in Figure 1.2.

2. (*easy*) Discover all of the possible meanings of the following sentences by giving a paraphrase of each interpretation. For each sentence, identify whether the different meanings arise from structural ambiguity, semantic ambiguity, or pragmatic ambiguity.

   a. Time flies like an arrow.
   b. He drew one card.
   c. Mr. Spock was charged with illegal alien recruitment.
   d. He crushed the key to my heart.

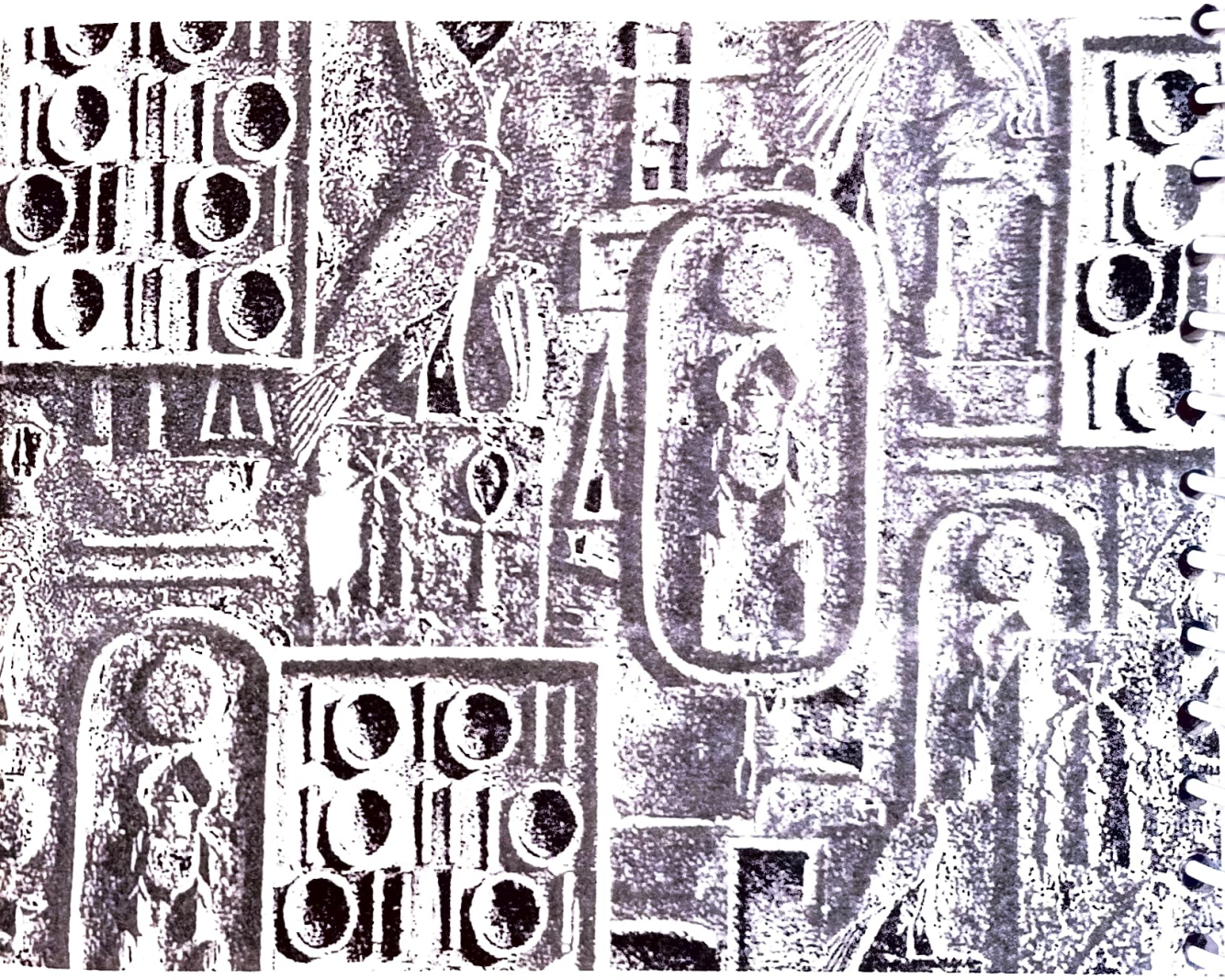
3. (*easy*) Classify these sentences along each of the following dimensions, given that the person uttering the sentence is responding to a complaint that the car is too cold: (i) syntactically correct or not; (ii) semantically correct or not; (iii) pragmatically correct or not.

   a. The heater are on.
   b. The tires are brand new.
   c. Too many windows eat the stew.

4. (*medium*) Implement an ELIZA program that can use the rules that you developed in Exercise 1 and run it for that dialogue. Without adding any more rules, what does your program do on the next few utterances in the conversation in Figure 1.2? How does the program do if you run it in a different context—say, a casual conversation at a bar?

PART I

Syntactic Processing



As discussed in the introduction, this book divides the task of understanding sentences into three stages. Part I of the book discusses the first stage, syntactic processing. The goal of syntactic processing is to determine the structural components of sentences. It determines, for instance, how a sentence is broken down into phrases, how those phrases are broken down into sub-phrases, and so on, all the way down to the actual structure of the words used. These structural relationships are crucial for determining the meaning of sentences using the techniques described in Parts II and III.

There are two major issues discussed in Part I. The first issue concerns the formalism that is used to specify what sentences are possible in a language. This information is specified by a set of rules called a grammar. We will be concerned both with the general issue of what constitutes good formalisms for writing grammars for natural languages, and with the specific issue of what grammatical rules provide a good account of English syntax. The second issue concerns how to determine the structure of a given sentence once you know the grammar for the language. This process is called parsing. There are many different algorithms for parsing, and this book will consider a sampling of the techniques that are most influential in the field.

Chapter 2 provides a basic background to English syntax for the reader who has not studied linguistics. It introduces the key concepts and distinctions that are common to virtually all syntactic theories. Chapter 3 introduces several formalisms that are in common use for specifying grammars and describes the basic parsing algorithms in detail. Chapter 4 introduces the idea of features, which extend the basic grammatical formalisms and allow many aspects of natural languages to be captured concisely. Chapter 5 then describes some of the more difficult aspects of natural languages, especially the treatment of questions, relative clauses, and other forms of movement phenomena. It shows how the feature systems can be extended so that they can handle these complex sentences. Chapter 6 discusses issues relating to ambiguity resolution. Some techniques are aimed at developing more efficient representations for storing multiple interpretations, while others are aimed at using local information to choose between alternative interpretations while the parsing is in progress. Finally, Chapter 7 discusses a relatively new area of research that uses statistical information derived from analyzing large databases of sentences. This information can be used to identify the most likely classes for ambiguous words and the most likely structural analyses for structurally ambiguous sentences.

CHAPTER

Linguistic Background: An Outline of English Syntax

This chapter provides background material on the basic structure of English syntax for those who have not taken any linguistics courses. It reviews the major phrase categories and identifies their most important subparts. Along the way all the basic word categories used in the book are introduced. While the only structures discussed are those for English, much of what is said applies to nearly all other European languages as well. The reader who has some background in linguistics can quickly skim this chapter, as it does not address any computational issues. You will probably want to use this chapter as a reference source as you work through the rest of the chapters in Part I.

Section 2.1 describes issues related to words and word classes. Section 2.2 describes simple noun phrases, which are then used in Section 2.3 to describe simple verb phrases. Section 2.4 considers complex noun phrases that include embedded sentences such as relative clauses. The remaining sections briefly cover other types of phrases: adjective phrases in Section 2.5 and adverbial phrases in Section 2.6.

2.1 Words

At first glance the most basic unit of linguistic structure appears to be the word. The word, though, is far from the fundamental element of study in linguistics; it is already the result of a complex set of more primitive parts. The study of **morphology** concerns the construction of words from more basic components corresponding roughly to meaning units. There are two basic ways that new words are formed, traditionally classified as **inflectional** forms and **derivational** forms. Inflectional forms use a **root** form of a word and typically add a **suffix** so that the word appears in the appropriate form given the sentence. Verbs are the best examples of this in English. Each verb has a basic form that then is typically changed depending on the subject and the tense of the sentence. For example, the verb *sigh* will take suffixes such as *-s*, *-ing*, and *-ed* to create the verb forms *sighs*, *sighing*, and *sighed*, respectively. These new words are all verbs and share the same basic meaning. Derivational morphology involves the derivation of new words from other forms. The new words may be in completely different categories from their subparts. For example, the noun *friend* is made into the adjective *friendly* by adding the suffix *-ly*. A more complex derivation would allow you to derive the noun *friendliness* from the adjective form. There are many interesting issues concerned with how words are derived and how the choice of word form is affected by the syntactic structure of the sentence that constrains it.

Traditionally, linguists classify words into different categories based on their uses. Two related areas of evidence are used to divide words into categories. The first area concerns the word's contribution to the meaning of the phrase that contains it, and the second area concerns the actual syntactic structures in which the word may play a role. For example, you might posit the class **noun** as those words that can be used to identify the basic type of object, concept, or place being discussed, and **adjective** as those words that further qualify the object,

concept, or place. Thus *green* would be an adjective and *book* a noun, as shown in the phrases *the green book* and *green books*. But things are not so simple: *green* might play the role of a noun, as in *That green is lighter than the other*, and *book* might play the role of a modifier, as in *the book worm*. In fact, most nouns seem to be able to be used as a modifier in some situations. Perhaps the classes should be combined, since they overlap a great deal. But other forms of evidence exist. Consider what words could complete the sentence *It's so . . .* You might say *It's so green, It's so hot, It's so true,* and so on. Note that although *book* can be a modifier in *the book worm*, you cannot say **It's so book* about anything. Thus there are two classes of modifiers: adjective modifiers and noun modifiers.

Consider again the case where adjectives can be used as nouns, as in *the green*. Not all adjectives can be used in such a way. For example, the noun phrase *the hot* can be used, given a context where there are hot and cold plates, in a sentence such as *The hot are on the table.* But this refers to the hot plates; it cannot refer to hotness in the way the phrase *the green* refers to green. With this evidence you could subdivide adjectives into two subclasses—those that can also be used to describe a concept or quality directly, and those that cannot. Alternatively, however, you can simply say that *green* is ambiguous between being an adjective or a noun and, therefore, falls in both classes. Since *green* can behave like any other noun, the second solution seems the most direct.

Using similar arguments, we can identify four main classes of words in English that contribute to the meaning of sentences. These classes are nouns, adjectives, verbs, and adverbs. Sentences are built out of phrases centered on these four word classes. Of course, there are many other classes of words that are necessary to form sentences, such as articles, pronouns, prepositions, particles, quantifiers, conjunctions, and so on. But these classes are fixed in the sense that new words in these classes are rarely introduced into the language. New nouns, verbs, adjectives and adverbs, on the other hand, are regularly introduced into the language as it evolves. As a result, these classes are called the **open class words**, and the others are called the **closed class words**.

A word in any of the four open classes may be used to form the basis of a phrase. This word is called the **head** of the phrase and indicates the type of thing, activity, or quality that the phrase describes. For example, with noun phrases, the head word indicates the general classes of objects being described. The phrases

the dog
the mangy dog
the mangy dog at the pound

are all noun phrases that describe an object in the class of dogs. The first describes a member from the class of all dogs, the second an object from the class of mangy dogs, and the third an object from the class of mangy dogs that are at the pound. The word *dog* is the head of each of these phrases.

Noun Phrases	Verb Phrases
The **president** *of the company*	**looked** *up the chimney*
His **desire** *to succeed*	**believed** *that the world was flat*
Several **challenges** *from the opposing team*	**ate** *the pizza*
Adjective Phrases	**Adverbial Phrases**
easy *to assemble*	**rapidly** *like a bat*
happy *that he'd won the prize*	**intermittently** *throughout the day*
angry *as a hippo*	**inside** *the house*

Figure 2.1 Examples of heads and complements

Similarly, the adjective phrases

hungry
very hungry
hungry as a horse

all describe the quality of hunger. In each case the word *hungry* is the head.

In some cases a phrase may consist of a single head. For example, the word *sand* can be a noun phrase, *hungry* can be an adjective phrase, and *walked* can be a verb phrase. In many other cases the head requires additional phrases following it to express the desired meaning. For example, the verb *put* cannot form a verb phrase in isolation; thus the following words do not form a meaningful sentence:

*Jack put.

To be meaningful, the verb *put* must be followed by a noun phrase and a phrase describing a location, as in the verb phrase *put the dog in the house*. The phrase or set of phrases needed to complete the meaning of such a head is called the **complement** of the head. In the preceding phrase *put* is the head and *the dog in the house* is the complement. Heads of all the major classes may require complements. Figure 2.1 gives some examples of phrases, with the head indicated by boldface and the complements by italics. In the remainder of this chapter, we will look at these different types of phrases in more detail and see how they are structured and how they contribute to the meaning of sentences.

2.2 The Elements of Simple Noun Phrases

Noun phrases (NPs) are used to refer to things: objects, places, concepts, events, qualities, and so on. The simplest NP consists of a single pronoun: *he, she, they, you, me, it, I,* and so on. Pronouns can refer to physical objects, as in the sentence

It hid under the rug.

to events, as in the sentence

Once I opened the door, I regretted *it* for months.

and to qualities, as in the sentence

He was so angry, but he didn't show *it*.

Pronouns do not take any modifiers except in rare forms, as in the sentence

He who hesitates is lost.

Another basic form of noun phrase consists of a **name** or **proper noun**, such as *John* or *Rochester*. These nouns appear in capitalized form in carefully written English. Names may also consist of multiple words, as in the *New York Times* and *Stratford-on-Avon*.

Excluding pronouns and proper names, the head of a noun phrase is usually a common noun. Nouns divide into two main classes:

count nouns—nouns that describe specific objects or sets of objects.
mass nouns—nouns that describe composites or substances.

Count nouns acquired their name because they can be counted. There may be one *dog* or many *dogs*, one *book* or several *books*, one *crowd* or several *crowds*. If a single count noun is used to describe a whole class of objects, it must be in its plural form. Thus you can say *Dogs are friendly* but not **Dog is friendly*.

Mass nouns cannot be counted. There may be *some water*, *some wheat*, or *some sand*. If you try to count with a mass noun, you change the meaning. For example, *some wheat* refers to a portion of some quantity of wheat, whereas *one wheat* is a single type of wheat rather than a single grain of wheat. A mass noun can be used to describe a whole class of material without using a plural form. Thus you say *Water is necessary for life*, not **Waters are necessary for life*.

In addition to a head, a noun phrase may contain **specifiers** and **qualifiers** preceding the head. The qualifiers further describe the general class of objects identified by the head, while the specifiers indicate how many such objects are being described, as well as how the objects being described relate to the speaker and hearer. Specifiers are constructed out of **ordinals** (such as *first* and *second*), **cardinals** (such as *one* and *two*), and **determiners**. Determiners can be subdivided into the following general classes:

articles—the words *the*, *a*, and *an*.
demonstratives—words such as *this*, *that*, *these*, and *those*.
possessives—noun phrases followed by the suffix *'s*, such as *John's* and *the fat man's*, as well as possessive pronouns, such as *her*, *my*, and *whose*.
wh-determiners—words used in questions, such as *which* and *what*.
quantifying determiners—words such as *some*, *every*, *most*, *no*, *any*, *both*, and *half*.

Number	First Person	Second Person	Third Person
singular	I	you	he (masculine) she (feminine) it (neuter)
plural	we	you	they

Figure 2.2 Pronoun system (as subject)

Number	First Person	Second Person	Third Person
singular	my	your	his, her, its
plural	our	your	their

Figure 2.3 Pronoun system (possessives)

A simple noun phrase may have at most one determiner, one ordinal, and one cardinal. It is possible to have all three, as in *the first three contestants.* An exception to this rule exists with a few quantifying determiners such as *many few, several,* and *little.* These words can be preceded by an article, yielding noun phrases such as *the few songs we knew.* Using this evidence, you could subcategorize the quantifying determiners into those that allow this and those that don't, but the present coarse categorization is fine for our purposes at this time.

The qualifiers in a noun phrase occur after the specifiers (if any) and before the head. They consist of adjectives and nouns being used as modifiers. The following are more precise definitions:

adjectives—words that attribute qualities to objects yet do not refer to the qualities themselves (for example, *angry* is an adjective that attributes the quality of anger to something).

noun modifiers—mass or count nouns used to modify another noun, as in *the cook book* or *the ceiling paint can.*

Before moving on to other structures, consider the different inflectional forms that nouns take and how they are realized in English. Two forms of nouns—the singular and plural forms—have already been mentioned. Pronouns take forms based on **person** (first, second, and third) and **gender** (masculine, feminine, and neuter). Each of these distinctions reflects a systematic analysis that is almost wholly explicit in some languages, such as Latin, while implicit in others. In French, for example, nouns are classified by their gender. In English many of these distinctions are not explicitly marked except in a few cases. The pronouns provide the best example of this. They distinguish **number**, **person**, **gender,** and **case** (that is, whether they are used as possessive, subject, or object), as shown in Figures 2.2 through 2.4.

Number	First Person	Second Person	Third Person
singular	me	you	him her it
plural	us	you	them

Figure 2.4 Pronoun system (as object)

Mood	Example
declarative (or assertion)	The cat is sleeping.
yes/no question	Is the cat sleeping?
wh-question	What is sleeping? or Which cat is sleeping?
imperative (or command)	Shoot the cat!

Figure 2.5 Basic moods of sentences

Form	Examples	Example Uses
base	hit, cry, go, be	*Hit* the ball! I want to *go*.
simple present	hit, cries, go, am	The dog *cries* every day. I *am* thirsty.
simple past	hit, cried, went, was	I *was* thirsty. I *went* to the store.
present participle	hitting, crying, going, being	I'm *going* to the store. *Being* the last in line aggravates me.
past participle	hit, cried, gone, been	I've *been* there before. The cake was *gone*.

Figure 2.6 The five verb forms

2.3 Verb Phrases and Simple Sentences

While an NP is used to refer to things, a sentence (S) is used to assert, query, or command. You may assert that some sentence is true, ask whether a sentence is true, or command someone to do something described in the sentence. The way a sentence is used is called its **mood**. Figure 2.5 shows four basic sentence moods.

A simple declarative sentence consists of an NP, the subject, followed by a verb phrase (VP), the predicate. A simple VP may consist of some adverbial modifiers followed by the head verb and its complements. Every verb must appear in one of the five possible forms shown in Figure 2.6.

Tense	The Verb Sequence	Example
simple present	simple present	He walks to the store.
simple past	simple past	He walked to the store.
simple future	*will* + infinitive	He will walk to the store.
present perfect	*have* in present + past participle	He has walked to the store.
future perfect	*will* + *have* in infinitive + past participle	I will have walked to the store.
past perfect (or pluperfect)	*have* in past + past participle	I had walked to the store.

Figure 2.7 The basic tenses

Tense	Structure	Example
present progressive	*be* in present + present participle	He is walking.
past progressive	*be* in past + present participle	He was walking.
future progressive	*will* + *be* in infinitive + present participle	He will be walking.
present perfect progressive	*have* in present + *be* in past participle + present participle	He has been walking.
future perfect progressive	*will* + *have* in present + *be* as past participle + present participle	He will have been walking.
past perfect progressive	*have* in past + *be* in past participle + present participle	He had been walking.

Figure 2.8 The progressive tenses

Verbs can be divided into several different classes: the **auxiliary verbs**, such as *be, do,* and *have;* the **modal verbs**, such as *will, can,* and *could;* and the **main verbs**, such as *eat, ran,* and *believe.* The auxiliary and modal verbs usually take a verb phrase as a complement, which produces a sequence of verbs, each the head of its own verb phrase. These sequences are used to form sentences with different tenses.

The **tense system** identifies when the proposition described in the sentence is said to be true. The tense system is complex; only the basic forms are outlined in Figure 2.7. In addition, verbs may be in the **progressive tense.** Corresponding to the tenses listed in Figure 2.7 are the progressive tenses shown in Figure 2.8.

	First	Second	Third
Singular	I *am* I *walk*	you *are* you *walk*	he *is* she *walks*
Plural	we *are* we *walk*	you *are* you *walk*	they *are* they *walk*

Figure 2.9 Person/number forms of verbs

Each progressive tense is formed by the normal tense construction of the verb *be* followed by a present participle.

Verb groups also encode person and number information in the first word in the verb group. The person and number must agree with the noun phrase that is the subject of the verb phrase. Some verbs distinguish nearly all the possibilities, but most verbs distinguish only the third person singular (by adding an *-s* suffix). Some examples are shown in Figure 2.9.

Transitivity and Passives

The last verb in a verb sequence is called the **main verb**, and is drawn from the open class of verbs. Depending on the verb, a wide variety of complement structures are allowed. For example, certain verbs may stand alone with no complement. These are called **intransitive** verbs and include examples such as *laugh* (for example, *Jack laughed*) and *run* (for example, *He will have been running*). Another common complement form requires a noun phrase to follow the verb. These are called **transitive** verbs and include verbs such as *find* (for example, *Jack found a key*). Notice that *find* cannot be intransitive (for example, **Jack found* is not a reasonable sentence), whereas *laugh* cannot be transitive (for example, **Jack laughed a key* is not a reasonable sentence). A verb like *run*, on the other hand, can be transitive or intransitive, but the meaning of the verb is different in each case (for example, *Jack ran* vs. *Jack ran the machine*).

Transitive verbs allow another form of verb group called the **passive** form, which is constructed using a *be* auxiliary followed by the past participle. In the passive form the noun phrase that would usually be in the object position is used in the subject position, as can be seen by the examples in Figure 2.10. Note that tense is still carried by the initial verb in the verb group. Also, even though the first noun phrase semantically seems to be the object of the verb in passive sentences, it is syntactically the subject. This can be seen by checking the pronoun forms. For example, *I was hit* is correct, not **Me was hit.* Furthermore, the tense and number agreement is between the verb and the syntactic subject. Thus you say *I was hit by them*, not **I were hit by them.*

Some verbs allow two noun phrases to follow them in a sentence; for example, *Jack gave Sue a book* or *Jack found me a key*. In such sentences the

Active Sentence	Related Passive Sentence
Jack saw the ball.	The ball was seen by Jack.
I will find the clue.	The clue will be found by me.
Jack hit me.	I was hit by Jack.

Figure 2.10 Active sentences with corresponding passive sentences

second NP corresponds to the object NP outlined earlier and is sometimes called the **direct object**. The other NP is called the **indirect object**. Generally, such sentences have an equivalent sentence where the indirect object appears with a preposition, as in *Jack gave a book to Sue* or *Jack found a key for me.*

Particles

Some verb forms are constructed from a verb and an additional word called a **particle**. Particles generally overlap with the class of prepositions considered in the next section. Some examples are *up, out, over,* and *in.* With verbs such as *look*, *take*, or *put*, you can construct many different verbs by combining the verb with a particle (for example, *look up, look out, look over,* and so on). In some sentences the difference between a particle and a preposition results in two different readings for the same sentence. For example, *look over the paper* would mean reading the paper, if you consider *over* a particle (the verb is *look over*). In contrast, the same sentence would mean looking at something else behind or above the paper, if you consider *over* a preposition (the verb is *look*).

You can make a sharp distinction between particles and prepositions when the object of the verb is a pronoun. With a verb-particle sentence, the pronoun must precede the particle, as in *I looked it up.* With the prepositional reading, the pronoun follows the preposition, as in *I looked up it.* Particles also may follow the object NP. Thus you can say *I gave up the game to Mary* or *I gave the game up to Mary.* This is not allowed with prepositions; for example, you cannot say **I climbed the ladder up.*

Clausal Complements

Many verbs allow clauses as complements. Clauses share most of the same properties of sentences and may have a subject, indicate tense, and occur in passivized forms. One common clause form consists of a sentence form preceded by the complementizer *that*, as in *that Jack ate the pizza.* This clause will be identified by the expression S[that], indicating a special subclass of S structures. This clause may appear as the complement of the verb *know*, as in *Sam knows that Jack ate the pizza.* The passive is possible, as in *Sam knows that the pizza was eaten by Jack.*

Another clause type involves the infinitive form of the verb. The VP[inf] clause is simply a VP starting in the infinitive form, as in the complement of the verb *wish* in *Jack wishes to eat the pizza*. An infinitive sentence S[inf] form is also possible where the subject is indicated by a *for* phrase, as in *Jack wishes for Sam to eat the pizza*.

Another important class of clauses are sentences with complementizers that are wh-words, such as *who, what, where, why, whether,* and *how many*. These question clauses, S[WH], can be used as a complement of verbs such as *know*, as in *Sam knows whether we went to the party* and *The police know who committed the crime*.

Prepositional Phrase Complements

Many verbs require complements that involve a specific prepositional phrase (PP). The verb *give* takes a complement consisting of an NP and a PP with the preposition *to*, as in *Jack gave the book to the library*. No other preposition can be used. Consider

> *Jack gave the book from the library. (OK only if *from the library* modifies book.)

In contrast, a verb like *put* can take any PP that describes a location, as in

> Jack put the book in the box.
> Jack put the book inside the box.
> Jack put the book by the door.

To account for this, we allow complement specifications that indicate prepositional phrases with particular prepositions. Thus the verb *give* would have a complement of the form NP+PP[to]. Similarly the verb *decide* would have a complement form NP+PP[about], and the verb *blame* would have a complement form NP+PP[on], as in *Jack blamed the accident on the police*.

Verbs such as *put*, which take any phrase that can describe a location (complement NP+Location), are also common in English. While locations are typically prepositional phrases, they also can be noun phrases, such as *home*, or particles, such as *back* or *here*. A distinction can be made between phrases that describe locations and phrases that describe a path of motion, although many location phrases can be interpreted either way. The distinction can be made in some cases, though. For instance, prepositional phrases beginning with *to* generally indicate a path of motion. Thus they cannot be used with a verb such as *put* that requires a location (for example, **I put the ball to the box*). This distinction will be explored further in Chapter 4.

Figure 2.11 summarizes many of the verb complement structures found in English. A full list would contain over 40 different forms. Note that while the examples typically use a different verb for each form, most verbs will allow several different complement structures.

Verb	Complement Structure	Example
laugh	Empty (intransitive)	Jack laughed.
find	NP (transitive)	Jack found a key.
give	NP+NP (bitransitive)	Jack gave Sue the paper.
give	NP+PP[to]	Jack gave the book to the library.
reside	Location phrase	Jack resides in Rochester
put	NP+Location phrase	Jack put the book inside.
speak	PP[with]+PP[about]	Jack spoke with Sue about the book.
try	VP[to]	Jack tried to apologize.
tell	NP+VP[to]	Jack told the man to go.
wish	S[to]	Jack wished for the man to go.
keep	VP[ing]	Jack keeps hoping for the best.
catch	NP+VP[ing]	Jack caught Sam looking in his desk.
watch	NP+VP[base]	Jack watched Sam eat the pizza.
regret	S[that]	Jack regretted that he'd eaten the whole thing.
tell	NP+S[that]	Jack told Sue that he was sorry.
seem	ADJP	Jack seems unhappy in his new job.
think	NP+ADJP	Jack thinks Sue is happy in her job.
know	S[WH]	Jack knows where the money is.

Figure 2.11 Some common verb complement structures in English

2.4 Noun Phrases Revisited

Section 2.2 introduced simple noun phrases. This section considers more complex forms in which NPs contain sentences or verb phrases as subcomponents.

All the examples in Section 2.2 had heads that took the null complement. Many nouns, however, may take complements. Many of these fall into the class of complements that require a specific prepositional phrase. For example, the noun *love* has a complement form PP[of], as in *their love of France*, the noun *reliance* has the complement form PP[on], as in *his reliance on handouts*, and the noun *familiarity* has the complement form PP[with], as in *a familiarity with computers*.

Many nouns, such as *desire*, *reluctance*, and *research*, take an infinitive VP form as a complement, as in the noun phrases *his desire to release the guinea pig*, *a reluctance to open the case again*, and *the doctor's research to find a cure for cancer*. These nouns, in fact, can also take the S[inf] form, as in *my hope for John to open the case again*.

Noun phrases can also be built out of clauses, which were introduced in the last section as the complements for verbs. For example, a *that* clause (S[that]) can be used as the subject of a sentence, as in the sentence *That George had the ring was surprising*. Infinitive forms of verb phrases (VP[inf]) and sentences (S[inf]) can also function as noun phrases, as in the sentences *To own a car would be*

delightful and *For us to complete a project on time would be unprecedented.* In addition, the **gerundive** forms (VP[ing] and S[ing]) can also function as noun phrases, as in the sentences *Giving up the game was unfortunate* and *John's giving up the game caused a riot.*

Relative clauses involve sentence forms used as modifiers in noun phrases. These clauses are often introduced by **relative pronouns** such as *who, which, that,* and so on, as in

The man *who gave Bill the money* . . .
The rug *that George gave to Ernest* . . .
The man *whom George gave the money to* . . .

In each of these relative clauses, the embedded sentence is the same structure as a regular sentence except that one noun phrase is missing. If this missing NP is filled in with the NP that the sentence modifies, the result is a complete sentence that captures the same meaning as what was conveyed by the relative clause. The missing NPs in the preceding three sentences occur in the subject position, in the object position, and as object to a preposition, respectively. Deleting the relative pronoun and filling in the missing NP in each produces the following:

The man gave Bill the money.
George gave *the rug* to Ernest.
George gave the money to *the man.*

As was true earlier, relative clauses can be modified in the same ways as regular sentences. In particular, passive forms of the preceding sentences would be as follows:

Bill was given the money by the man.
The rug was given to Ernest by George.
The money was given to the man by George.

Correspondingly, these sentences could have relative clauses in the passive form as follows:

The man *Bill was given the money by* . . .
The rug *that was given to Ernest by George* . . .
The man whom *the money was given to by George* . . .

Notice that some relative clauses need not be introduced by a relative pronoun. Often the relative pronoun can be omitted, producing what is called a **base relative clause**, as in the NP *the man George gave the money to.* Yet another form deletes the relative pronoun and an auxiliary *be* form, creating a **reduced relative clause**, as in the NP *the man given the money,* which means the same as the NP *the man who was given the money.*

2.5 Adjective Phrases

You have already seen simple adjective phrases (ADJPs) consisting of a single adjective in several examples. More complex adjective phrases are also possible, as adjectives may take many of the same complement forms that occur with verbs. This includes specific prepositional phrases, as with the adjective *pleased*, which takes the complement form PP[with] (for example, *Jack was pleased with the prize*), or *angry* with the complement form PP[at] (for example, *Jack was angry at the committee*). *Angry* also may take an S[that] complement form, as in *Jack was angry that he was left behind.* Other adjectives take infinitive forms, such as the adjective *willing* with the complement form VP[inf], as in *Jack seemed willing to lead the chorus.*

These more complex adjective phrases are most commonly found as the complements of verbs such as *be* or *seem*, or following the head in a noun phrase. They generally cannot be used as modifiers preceding the heads of noun phrases (for example, consider **the angry at the committee man* vs. *the angry man* vs. *the man angry at the committee*).

Adjective phrases may also take a degree modifier preceding the head, as in the adjective phrase *very angry* or *somewhat fond of Mary.* More complex degree modifications are possible, as in *far too heavy* and *much more desperate.* Finally, certain constructs have degree modifiers that involve their own complement forms, as in *too stupid to come in out of the rain, so boring that everyone fell asleep,* and *as slow as a dead horse.*

2.6 Adverbial Phrases

You have already seen adverbs in use in several constructs, such as indicators of degree (for example, *very, rather, too*), and in location phrases (for example, *here, everywhere*). Other forms of adverbs indicate the manner in which something is done (for example, *slowly, hesitantly*), the time of something (for example, *now, yesterday*), or the frequency of something (for example, *frequently, rarely, never*).

Adverbs may occur in several different positions in sentences: in the sentence initial position (for example, ***Then***, *Jack will open the drawer*), in the verb sequence (for example, *Jack* ***then*** *will open the drawer, Jack will* ***then*** *open the drawer*), and in the sentence final position (for example, *Jack opened the drawer* ***then***). The exact restrictions on what adverb can go where, however, is quite idiosyncratic to the particular adverb.

In addition to these adverbs, adverbial modifiers can be constructed out of a wide range of constructs, such as prepositional phrases indicating, among other things, location (for example, *in the box*) or manner (for example, *in great haste*); noun phrases indicating, among other things, frequency (for example, *every day*); or clauses indicating, among other things, the time (for example, *when the bomb exploded*). Such adverbial phrases, however, usually cannot occur except in the sentence initial or sentence final position. For example, we can say *Every day*

John opens his drawer or *John opens his drawer every day*, but not **John every day opens his drawer*.

Because of the wide range of forms, it generally is more useful to consider adverbial phrases (ADVPs) by function rather than syntactic form. Thus we can consider manner, temporal, duration, location, degree, and frequency adverbial phrases each as its own form. We considered the location and degree forms earlier, so here we will consider some of the others.

Temporal adverbials occur in a wide range of forms: adverbial particles (for example, *now*), noun phrases (for example, *today, yesterday*), prepositional phrases (for example, *at noon, during the fight*), and clauses (for example, *when the clock struck noon, before the fight started*).

Frequency adverbials also can occur in a wide range of forms: particles (for example, *often*), noun phrases (for example, *every day*), prepositional phrases (for example, *at every party*), and clauses (for example, *every time that John comes for a visit*).

Duration adverbials appear most commonly as prepositional phrases (for example, *for three hours, about 20 feet*) and clauses (for example, *until the moon turns blue*).

Manner adverbials occur in a wide range of forms, including particles (for example, *slowly*), noun phrases (for example, *this way*), prepositional phrases (for example, *in great haste*), and clauses (for example, *by holding the embers at the end of a stick*).

In the analyses that follow, adverbials will most commonly occur as modifiers of the action or state described in a sentence. As such, an issue arises as to how to distinguish verb complements from adverbials. One distinction is that adverbial phrases are always optional. Thus you should be able to delete the adverbial and still have a sentence with approximately the same meaning (missing, obviously, the contribution of the adverbial). Consider the sentences

Jack put the box by the door.
Jack ate the pizza by the door.

In the first sentence the prepositional phrase is clearly a complement, since deleting it to produce **Jack put the box* results in a nonsensical utterance. On the other hand, deleting the phrase from the second sentence has only a minor effect: *Jack ate the pizza* is just a less general assertion about the same situation described by *Jack ate the pizza by the door.*

Summary

The major phrase structures of English have been introduced—namely, noun phrases, sentences, prepositional phrases, adjective phrases, and adverbial phrases. These will serve as the building blocks for the syntactic structures introduced in the following chapters.

Related Work and Further Readings

An excellent overview of English syntax is found in Baker (1989). The most comprehensive sources are books that attempt to describe the entire structure of English, such as Huddleston (1988), Quirk et al. (1972), and Leech and Svartvik (1975).

Exercises for Chapter 2

1. (*easy*) The text described several different example tests for distinguishing word classes. For example, nouns can occur in sentences of the form *I saw the* X, whereas adjectives can occur in sentences of the form *It's so* X. Give some additional tests to distinguish these forms and to distinguish between count nouns and mass nouns. State whether each of the following words can be used as an adjective, count noun, or mass noun. If the word is ambiguous, give all its possible uses.

milk, house, liquid, green, group, concept, airborne

2. (*easy*) Identify every major phrase (noun, verb, adjective, or adverbial phrases) in the following sentences. For each, indicate the head of the phrase and any complements of the head. Be sure to distinguish between complements and optional modifiers.

The man played his fiddle in the street.
The people dissatisfied with the verdict left the courtroom.

3. (*easy*) A very useful test for determining the syntactic role of words and phrases is the conjunction test. Conjunctions, such as *and* and *or*, tend to join together two phrases of the same type. For instance, you can conjoin nouns, as in *the man and woman*; noun phrases, as in *the angry men and the large dogs*; and adjective phrases, as in *the cow, angry and confused, broke the gate*. In each of the following sentences, identify the type of phrase being conjoined, and underline each phrase.

He was tired and hungrier than a herd of elephants.
We have never walked home or to the store from here.
The dog returned quickly and dropped the stick.

4. (*easy*) Explain in detail, using the terminology of this chapter, why each of the following sentences is ill formed. In particular, state what rule (given in this chapter) has been violated.

a. He barked the wrong tree up.
b. She turned waters into wine.
c. Don't take many all the cookies!
d. I feel floor today.
e. They all laughed the boy.

5. (*easy*) Classify the following verbs as being intransitive, transitive, or bitransitive (that is, it takes a two-NP complement). If the verb can be used in more than one of these forms, give each possible classification. Give an example sentence for each form to demonstrate your analysis.

   a. cry
   b. sing
   c. donate
   d. put

6. (*easy*) Using the verb *to be*, give example sentences that use the six basic tenses listed in Figure 2.7 and the six progressive tenses listed in Figure 2.8.

7. (*easy*) Using the verb *donate*, give examples of the passive form for each of the six basic tenses listed in Figure 2.7.

8. (*easy*) Classify the following verbs by specifying what different complement structures they allow, using the forms defined in Figure 2.11.

   give, know, assume, insert

   Give an example of an additional complement structure that is allowed by one of these verbs but not listed in the figure.

9. (*easy*) Find five verbs not discussed in this chapter that take an indirect object and, for each one, give a paraphrase of the same sentence using a prepositional phrase instead of the indirect object. Try for as wide a range as possible. Can you find one that cannot be paraphrased using either the preposition *to* or *for*?

10. (*medium*) Wh-questions are questions that use a class of words that includes *what, where, who, when, whose, which,* and *how*. For each of these words, give the syntactic categories (for example, verb, noun, noun group, adjective, quantifier, prepositional phrase, and so on) in which the words can be used. Justify each classification with some examples that demonstrate it. Use both positive and negative arguments as necessary (such as "it is one of these because . . .," or "it can't be one of these even though it looks like it might, because . . .").

CHAPTER

Grammars and Parsing

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

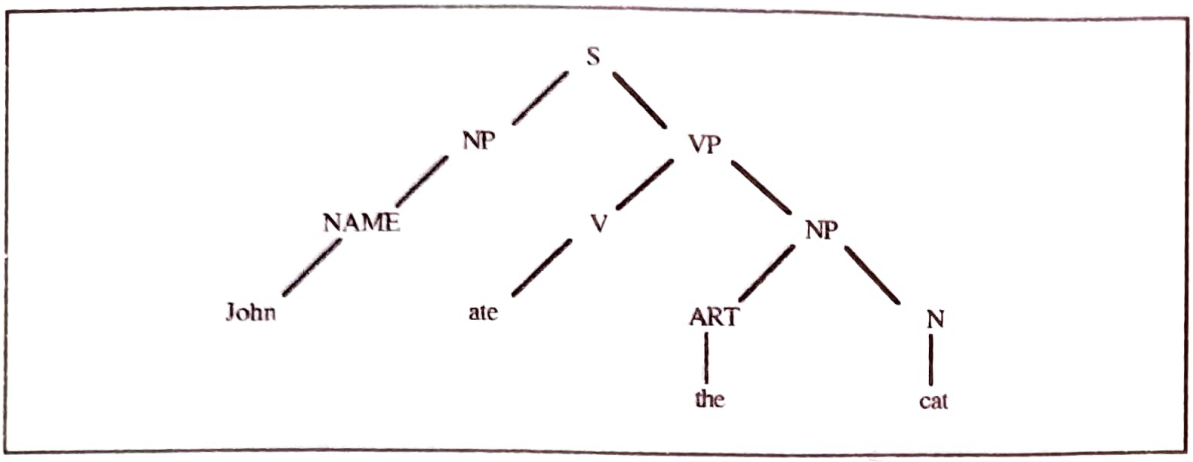


Figure 3.1 A tree representation of *John ate the cat*

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 performed to see if a new analysis is reasonable or not. Sometimes the analysis of a

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**).

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))

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.

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)		rewriting NP by rules 2 & 3
		((ART ADJ N VP) 1)	
4.	((N VP) 2)		matching ART with *the*
		((ART ADJ N VP) 1)	
5.	((VP) 3)		matching N with *dogs*
		((ART ADJ N VP) 1)	
6.	((V) 3)		rewriting VP by rules 5–8
		((V NP) 3)	
		((ART ADJ N VP) 1)	
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* $_4$

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.

Step	Current State	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* $_5$

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.

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.

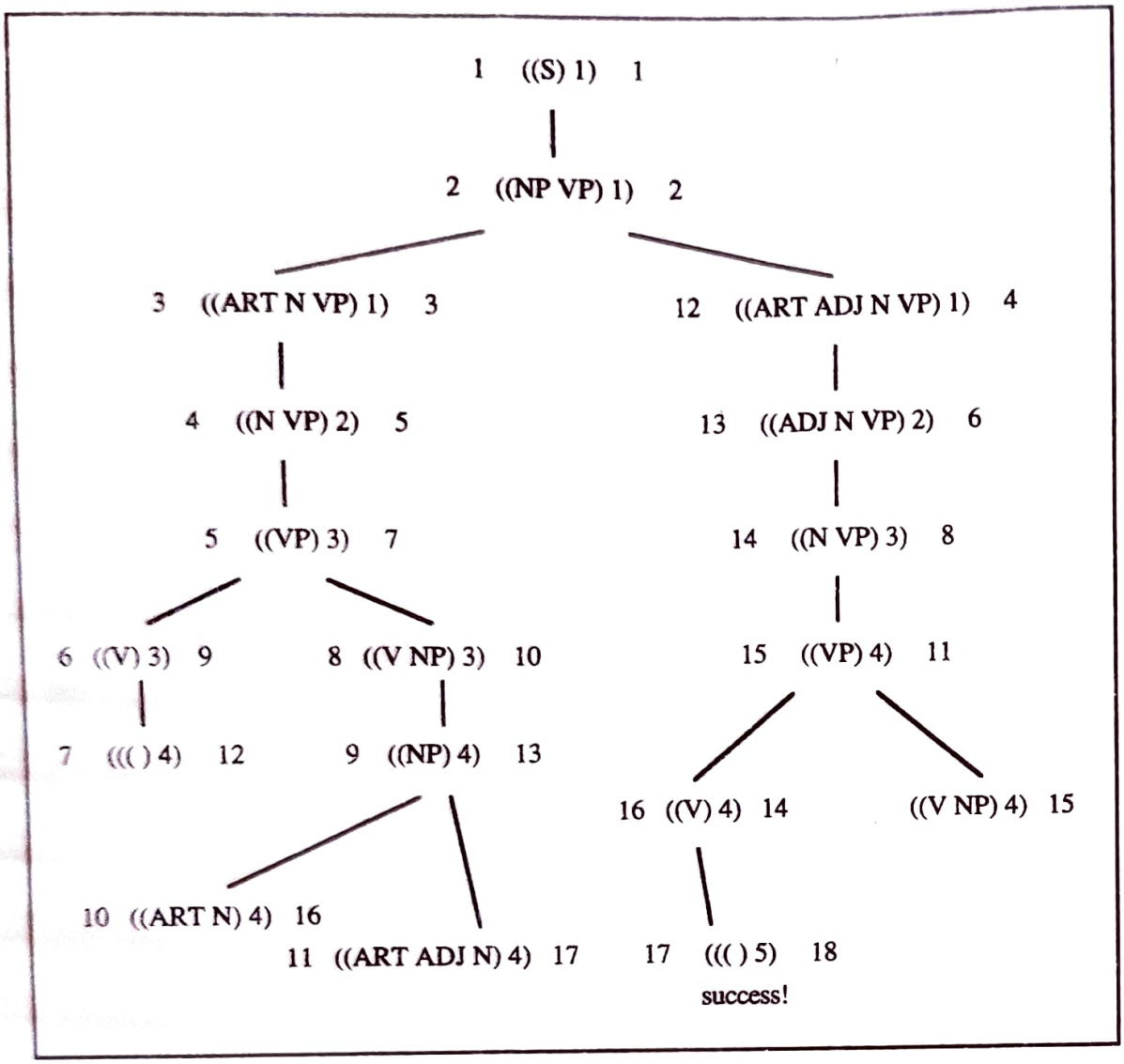


Figure 3.7 Search tree for two parse strategies (depth-first strategy on left; breadth-first on right)

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 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

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

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

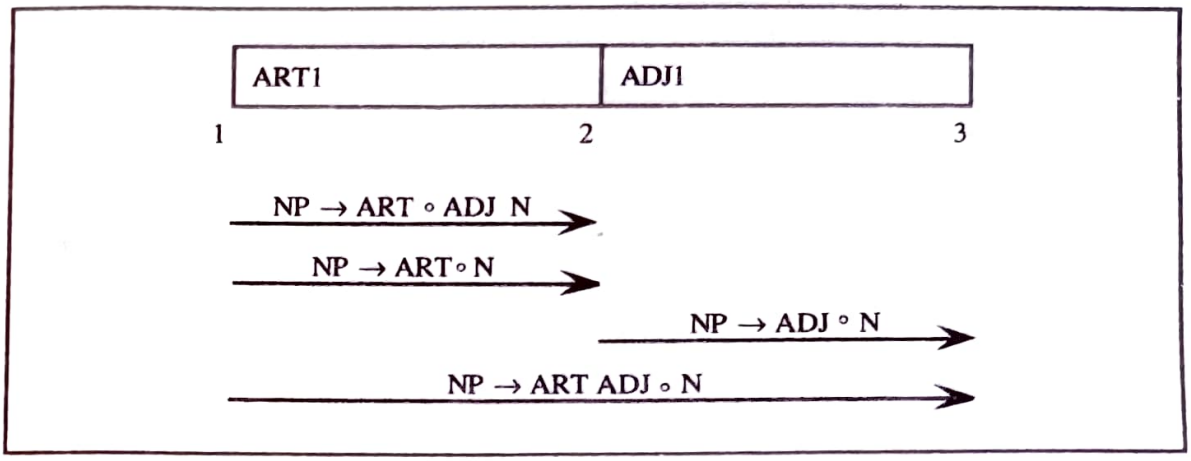


Figure 3.9 The chart after seeing an ADJ in position 2

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

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.

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:

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

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

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.

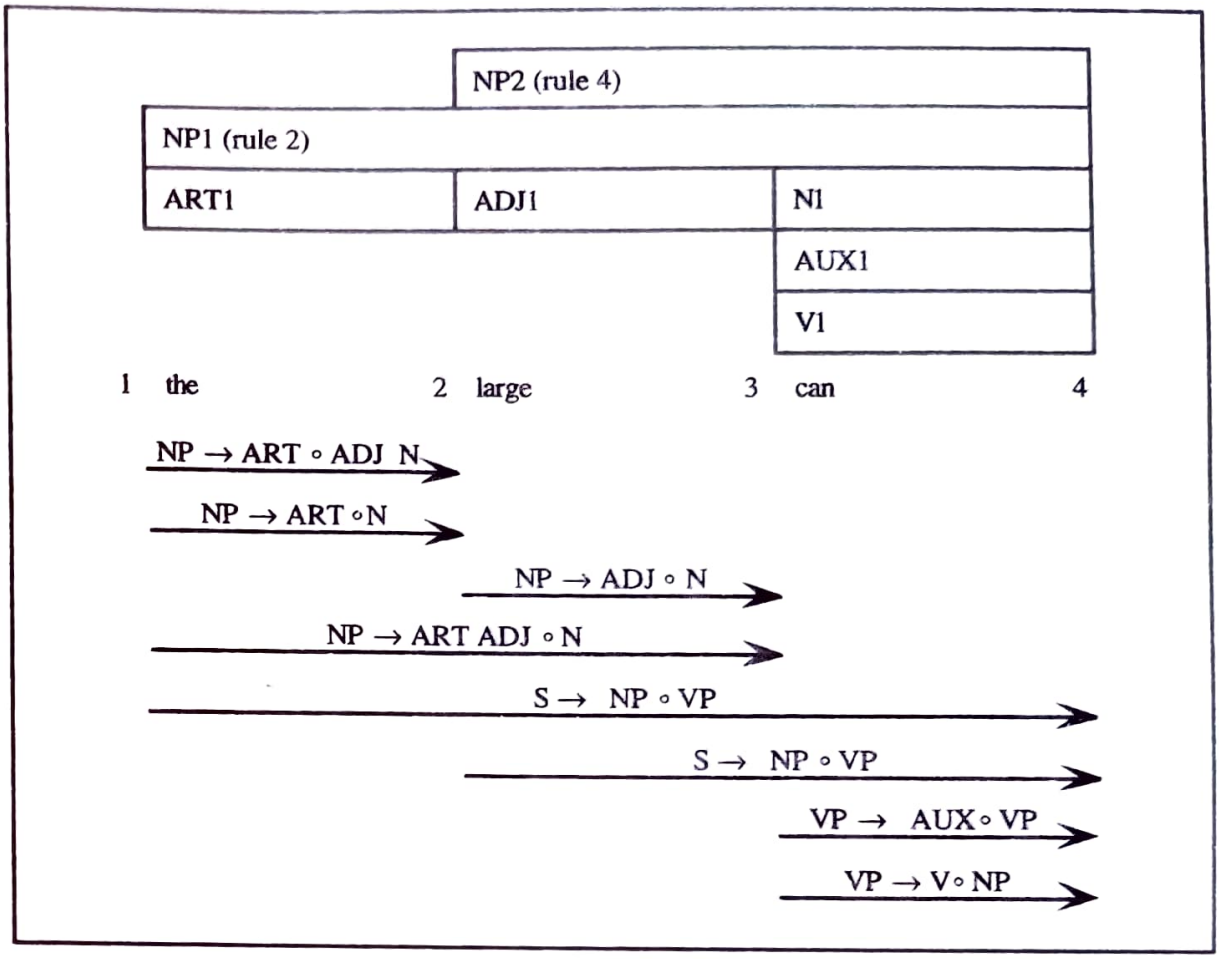


Figure 3.12 After parsing *the large can*

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.

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.

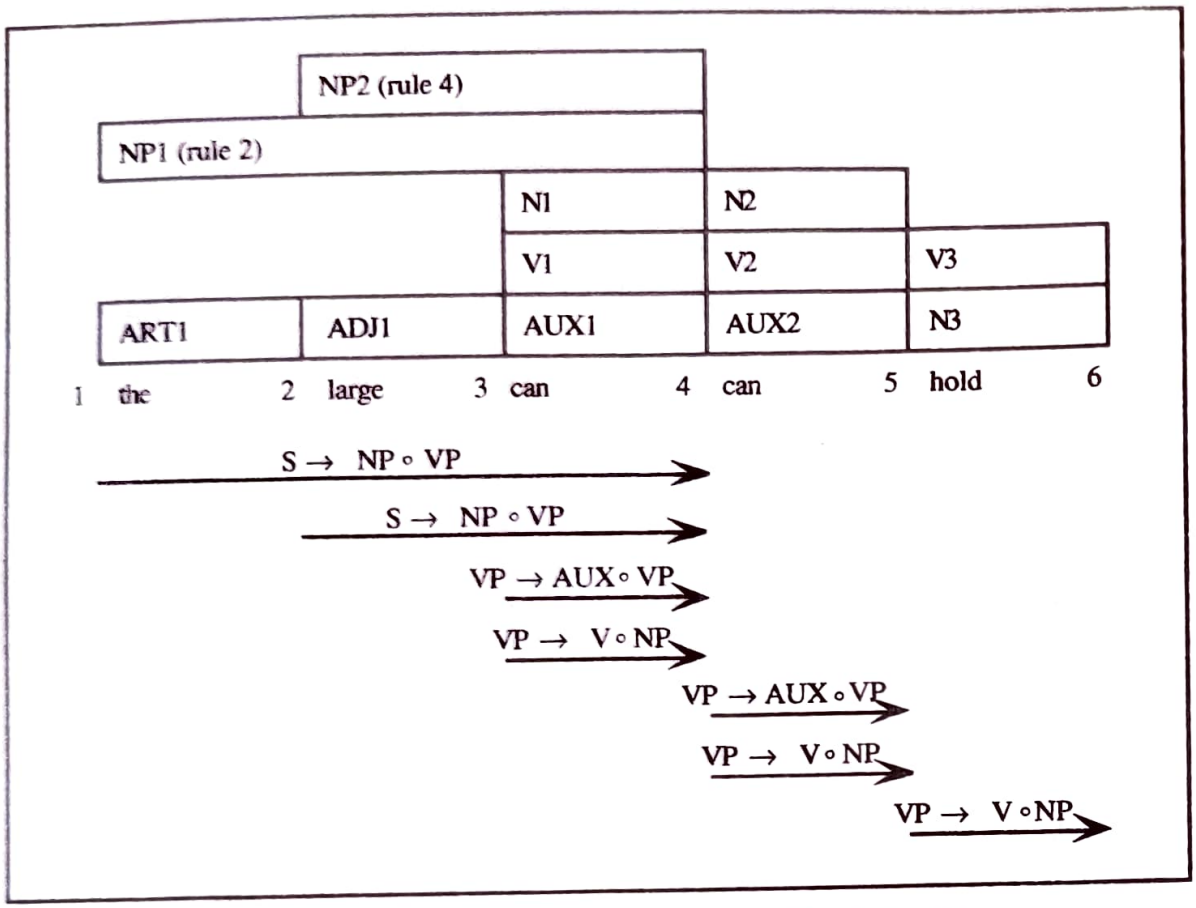


Figure 3.13 The chart after adding *hold*, omitting arcs generated for the first NP

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
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

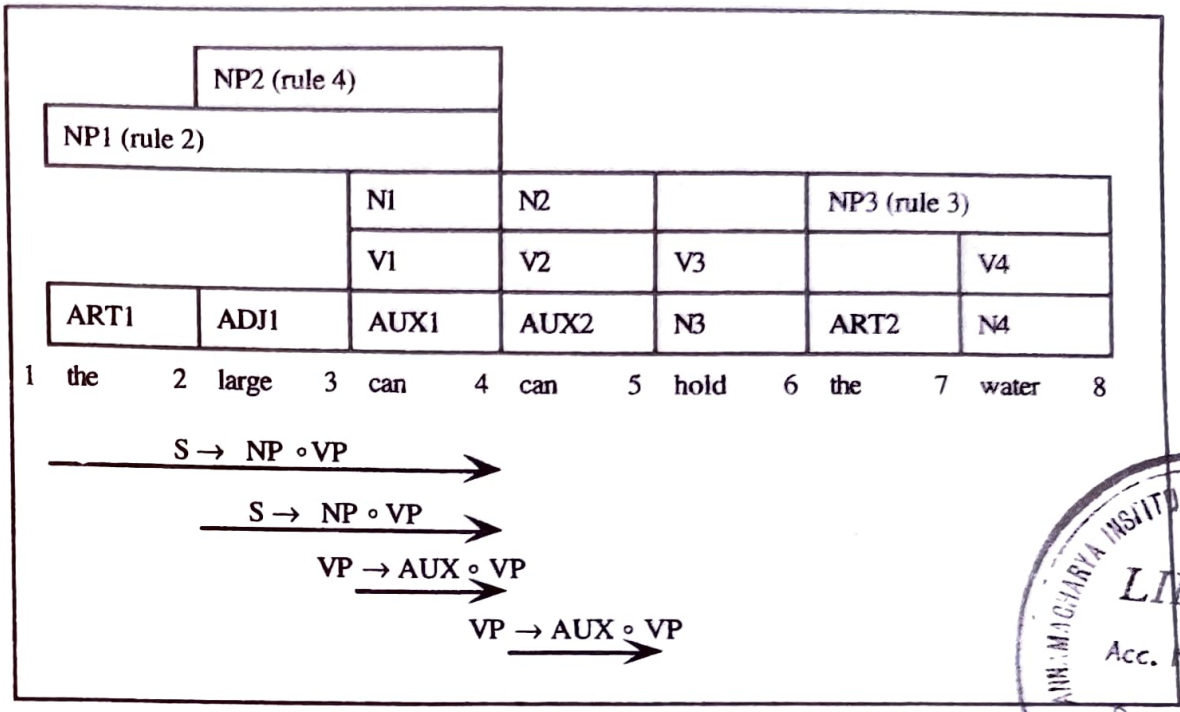


Figure 3.14 The chart after all the NPs are found, omitting all but the crucial active arcs

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,

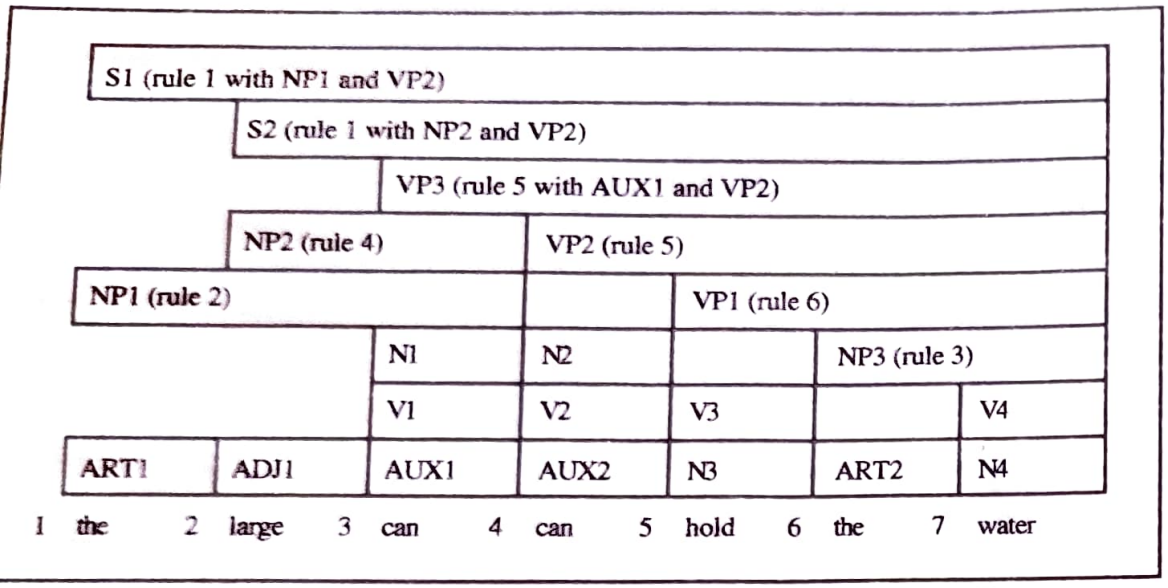


Figure 3.15 The final chart

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

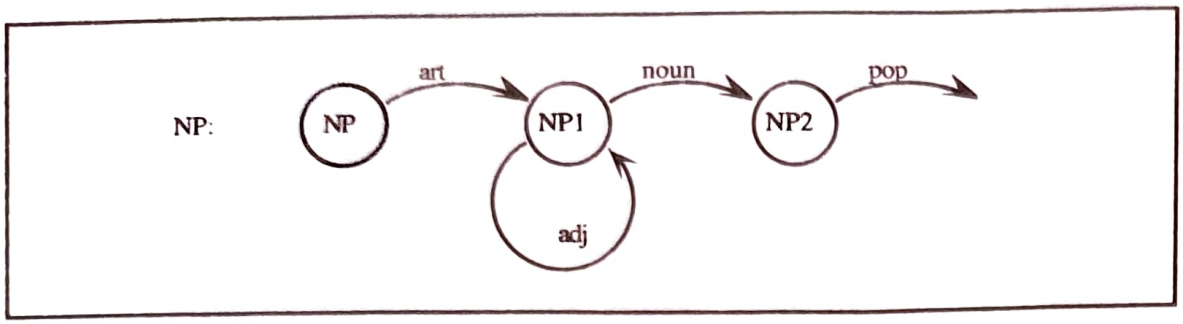
So far we have examined only one formalism for representing grammars, namely context-free rewrite rules. Here we consider another formalism that is useful in a wide range of applications. It is based on the notion of a **transition network** consisting of **nodes** and **labeled arcs**. One of the nodes 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

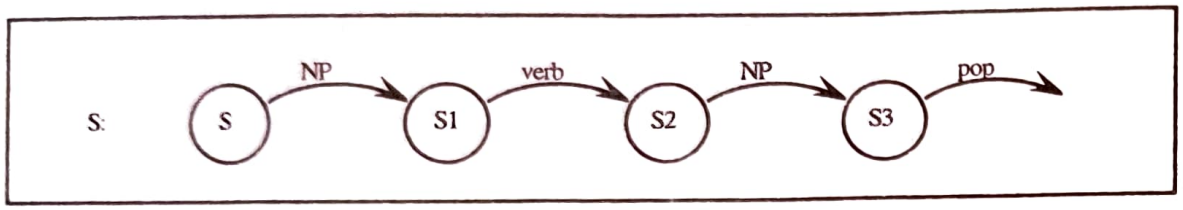
Consider parsing the NP *a purple cow* with this network. Starting at the node NP, you can follow the arc labeled art, since the current word is an article—namely, *a*. From node NP1 you can follow the arc labeled adj using the adjective *purple*, and finally, again from NP1, you can follow the arc labeled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

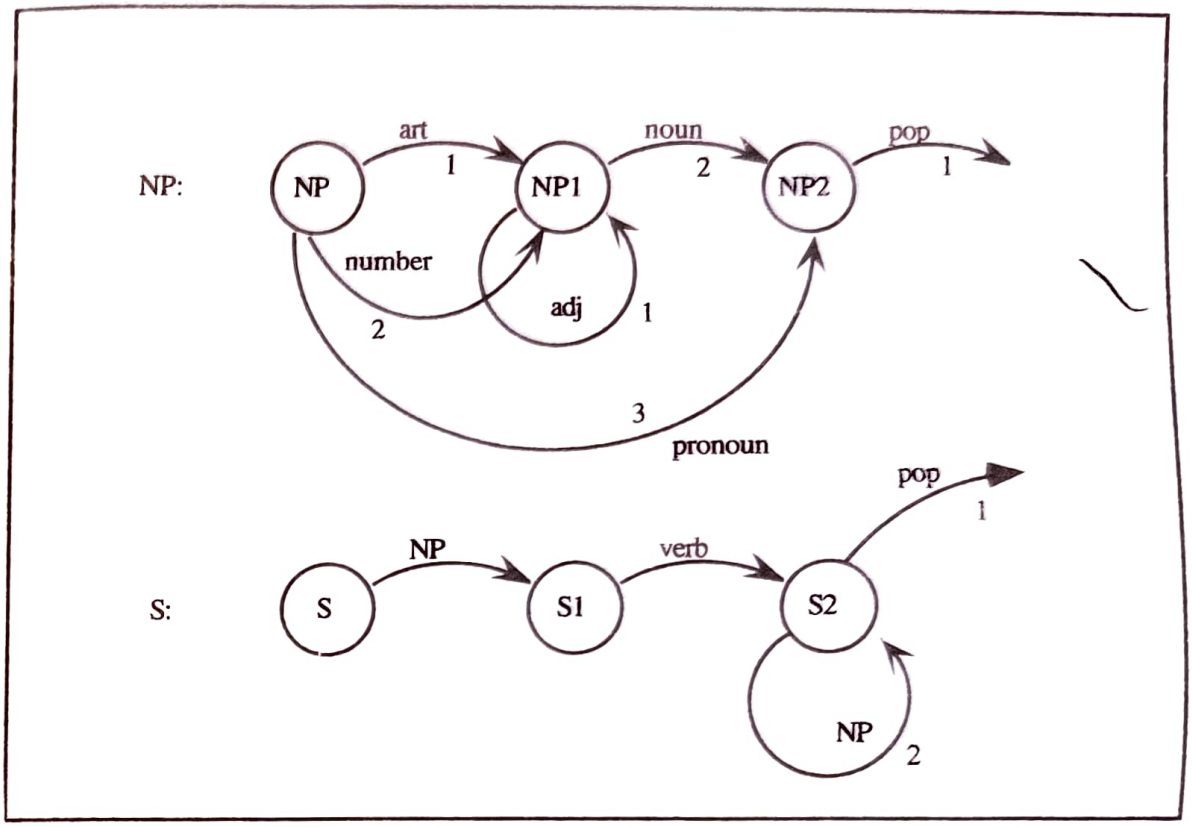
Figure 3.18 The arc labels for RTNs

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 $\rightarrow \circ$ NP VP is added. Then, active arcs for each rule that can derive an NP are added: NP $\rightarrow \circ$ ART ADJ N, NP $\rightarrow \circ$ 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.

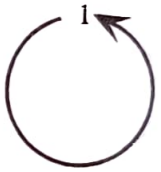
Top-Down Chart Parsing Algorithm

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



$S \rightarrow \circ NP\ VP$
$NP \rightarrow \circ ADJ\ N$
$NP \rightarrow \circ ART\ N$
$NP \rightarrow \circ ART\ ADJ\ N$

Figure 3.23 The initial chart

and $NP \rightarrow \circ ADJ\ N$ 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
        $NP \rightarrow ART \circ N$ from 1 to 2
        $NP \rightarrow ART \circ ADJ\ N$ from 1 to 2
Entering ADJ1 (*large*) from 2 to 3
    One arc can be extended
        $NP \rightarrow ART\ ADJ \circ N$ 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

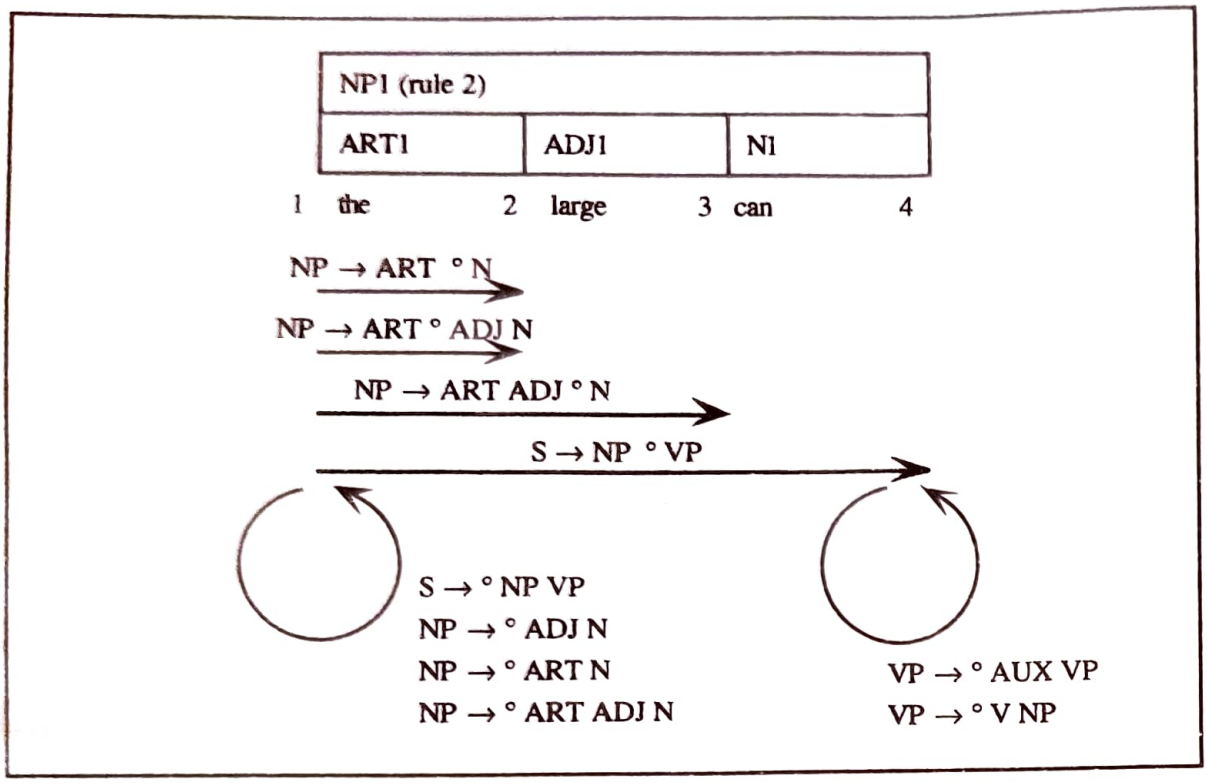


Figure 3.24 The chart after building the first NP

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), you see that the number of constituents generated has dropped from 21 to 13.

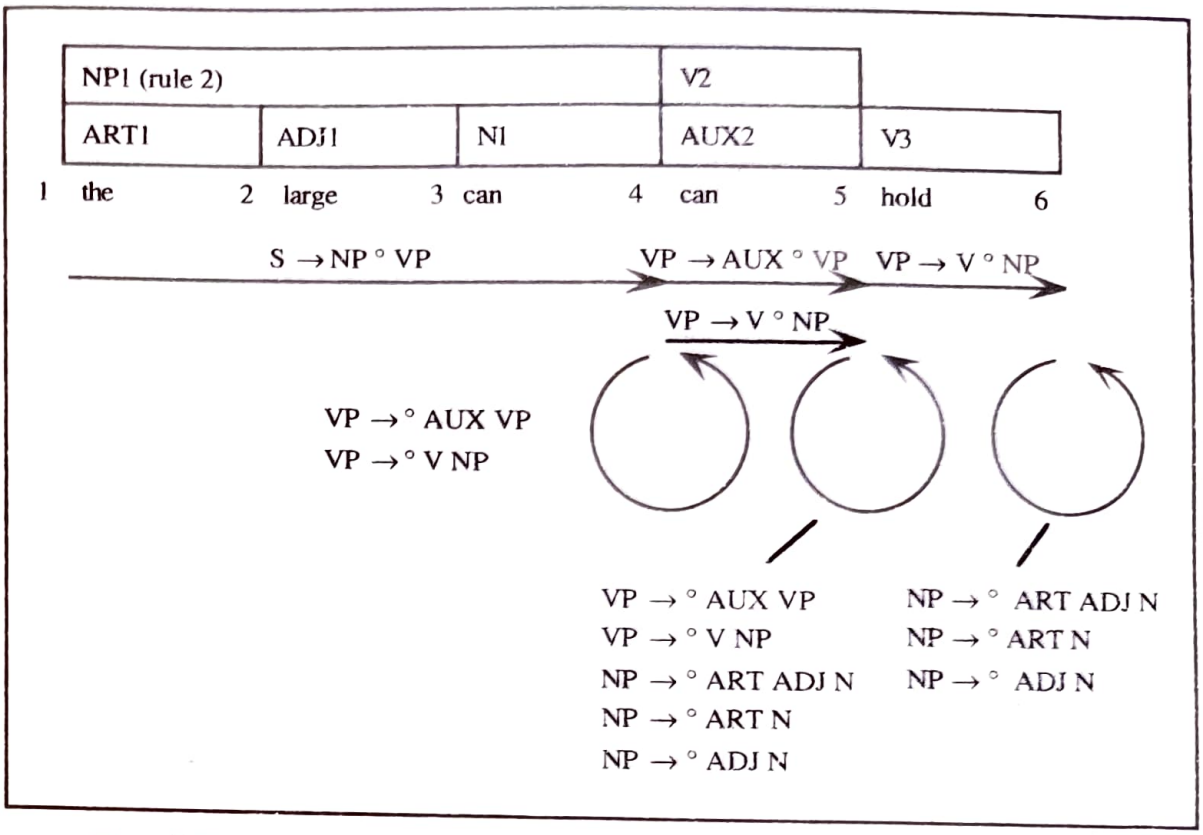


Figure 3.25 The chart after adding *hold*, omitting arcs generated for the first NP

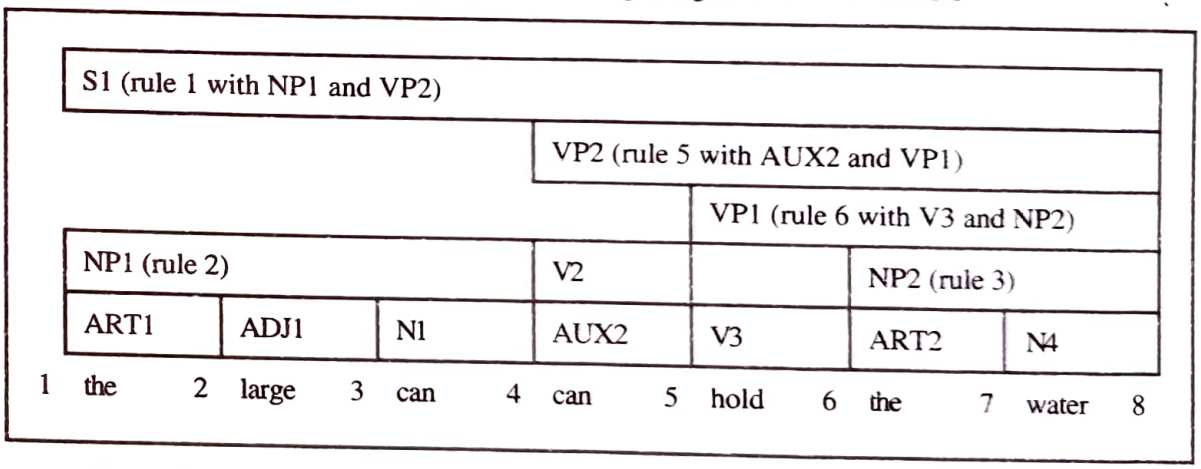


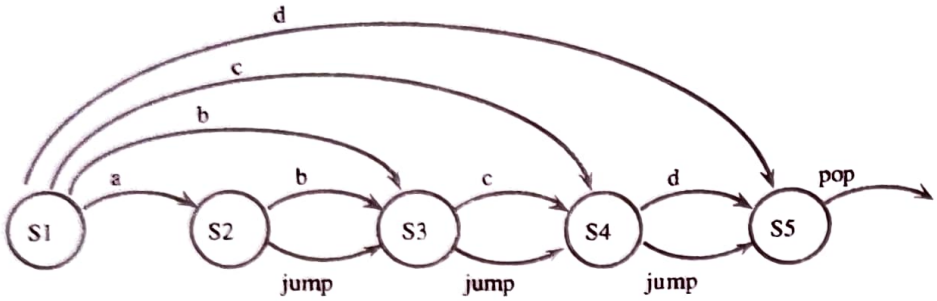
Figure 3.26 The final chart for the top-down filtering algorithm

While it is not a big difference here with such a simple grammar, the difference can be dramatic with a sizable grammar.

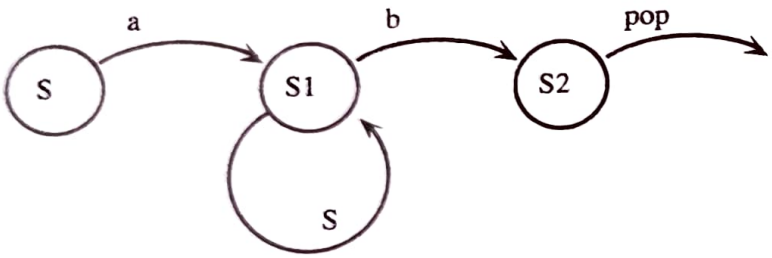
It turns out in the worst-case analysis that the top-down chart parser is not more efficient that the pure bottom-up chart parser. Both have a worst-case complexity of $K*n^3$ for a sentence of length n. In practice, however, the top-down method is considerably more efficient for any reasonable grammar.

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

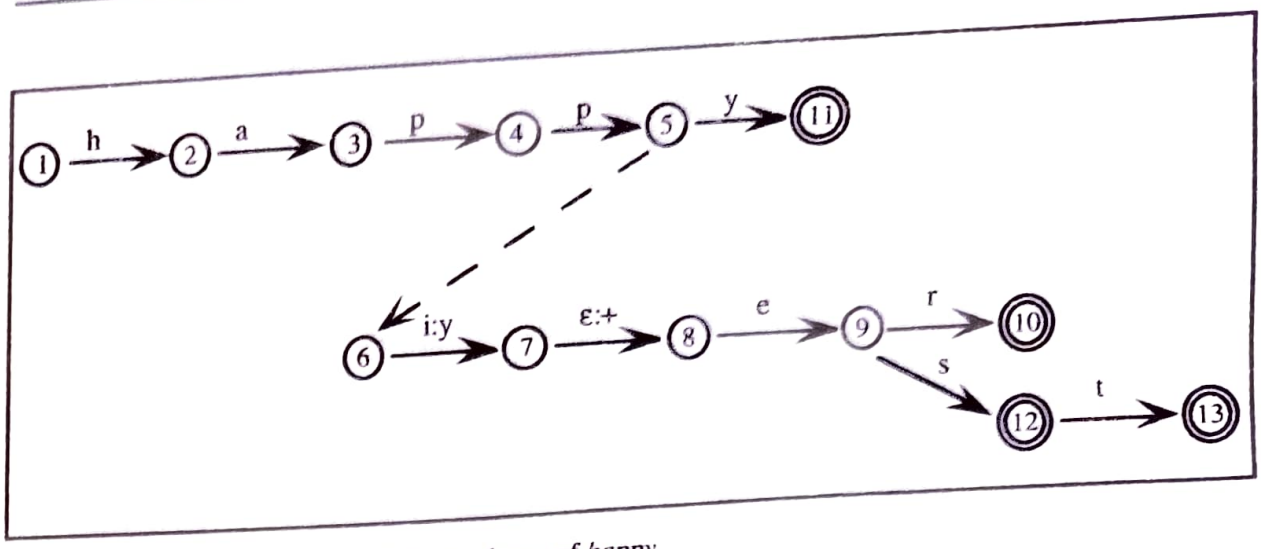


Figure 3.27 A simple FST for the forms of *happy*

ate) rather than four. Likewise the word *happiest* breaks down into the root form *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

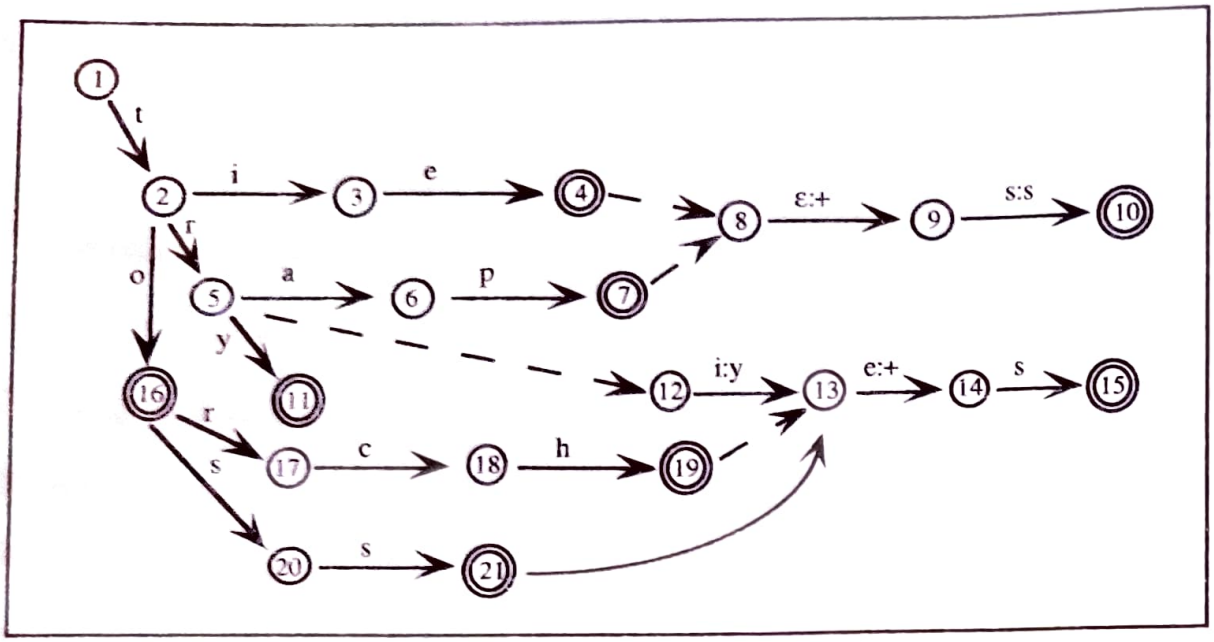


Figure 3.28 A fragment of an FST defining some nouns (singular and plural)

different suffixes need only be defined once, and all root forms that allow that 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:

$$S \rightarrow NP\ VP$$

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

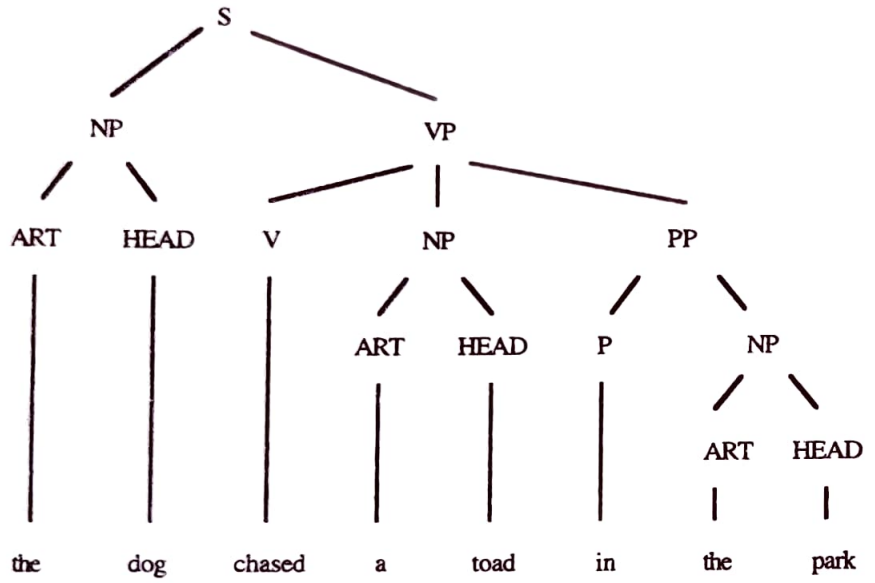
language that allows general morphological rules to be expressed concisely. A compiler can then be used to generate the appropriate networks for the system.

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.

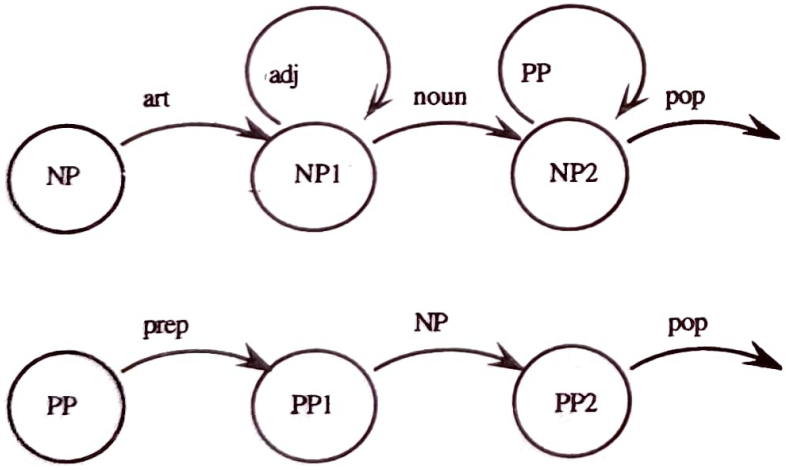
9. (*medium*) Consider the following CFG that generates sequences of letters:

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.

Context-free grammars provide the basis for most of the computational parsing mechanisms developed to date, but as they have been described so far, they would be very inconvenient for capturing natural languages. This chapter describes an extension to the basic context-free mechanism that defines constituents by a set of **features**. This extension allows aspects of natural language such as agreement and subcategorization to be handled in an intuitive and concise way.

Section 4.1 introduces the notion of feature systems and the generalization of context-free grammars to allow features. Section 4.2 then describes some useful feature systems for English that are typical of those in use in various grammars. Section 4.3 explores some issues in defining the lexicon and shows how using features makes the task considerably simpler. Section 4.4 describes a sample context-free grammar using features and introduces some conventions that simplify the process. Section 4.5 describes how to extend a chart parser to handle a grammar with features. The remaining sections, which are optional, describe how features are used in other grammatical formalisms and explore some more advanced material. Section 4.6 introduces augmented transition networks, which are a generalization of recursive transition networks with features, and Section 4.7 describes definite clause grammars based on PROLOG. Section 4.8 describes generalized feature systems and unification grammars.

4.1 Feature Systems and Augmented Grammars

In natural languages there are often agreement restrictions between words and phrases. For example, the NP *a men* is not correct English because the article *a* indicates a single object while the noun *men* indicates a plural object; the noun phrase does not satisfy the **number agreement** restriction of English. There are many other forms of agreement, including subject-verb agreement, gender agreement for pronouns, restrictions between the head of a phrase and the form of its complement, and so on. To handle such phenomena conveniently, the grammatical formalism is extended to allow constituents to have **features**. For example, we might define a feature NUMBER that may take a **value** of either *s* (for singular) or *p* (for plural), and we then might write an augmented CFG rule such as

NP → ART N only when NUMBER$_1$ agrees with NUMBER$_2$

This rule says that a legal noun phrase consists of an article followed by a noun, but only when the number feature of the first word agrees with the number feature of the second. This one rule is equivalent to two CFG rules that would use different terminal symbols for encoding singular and plural forms of all noun phrases, such as

NP-SING → ART-SING N-SING
NP-PLURAL → ART-PLURAL N-PLURAL

While the two approaches seem similar in ease-of-use in this one example, consider that all rules in the grammar that use an NP on the right-hand side would

now need to be duplicated to include a rule for NP-SING and a rule for NP-PLURAL, effectively doubling the size of the grammar. And handling additional features, such as person agreement, would double the size of the grammar again and again. Using features, the size of the augmented grammar remains the same as the original one yet accounts for agreement constraints.

To accomplish this, a constituent is defined as a **feature structure**—a mapping from features to values that defines the relevant properties of the constituent. In the examples in this book, feature names in formulas will be written in boldface. For example, a feature structure for a constituent ART1 that represents a particular use of the word *a* might be written as follows:

ART1: (**CAT** ART
ROOT a
NUMBER s)

This says it is a constituent in the category ART that has as its root the word *a* and is singular. Usually an abbreviation is used that gives the CAT value more prominence and provides an intuitive tie back to simple context-free grammars. In this abbreviated form, constituent ART1 would be written as

ART1: (ART **ROOT** a **NUMBER** s)

Feature structures can be used to represent larger constituents as well. To do this, feature structures themselves can occur as values. Special features based on the integers—1, 2, 3, and so on—will stand for the first subconstituent, second subconstituent, and so on, as needed. With this, the representation of the NP constituent for the phrase *a fish* could be

NP1: (NP **NUMBER** s
1 (ART **ROOT** a
NUMBER s)
2 (N **ROOT** fish
NUMBER s))

Note that this can also be viewed as a representation of a parse tree shown in Figure 4.1, where the subconstituent features 1 and 2 correspond to the subconstituent links in the tree.

The rules in an augmented grammar are stated in terms of feature structures rather than simple categories. Variables are allowed as feature values so that a rule can apply to a wide range of situations. For example, a rule for simple noun phrases would be as follows:

(NP **NUMBER** ?n) → (ART **NUMBER** ?n) (N **NUMBER** ?n)

This says that an NP constituent can consist of two subconstituents, the first being an ART and the second being an N, in which the NUMBER feature in all three constituents is identical. According to this rule, constituent NP1 given previously is a legal constituent. On the other hand, the constituent

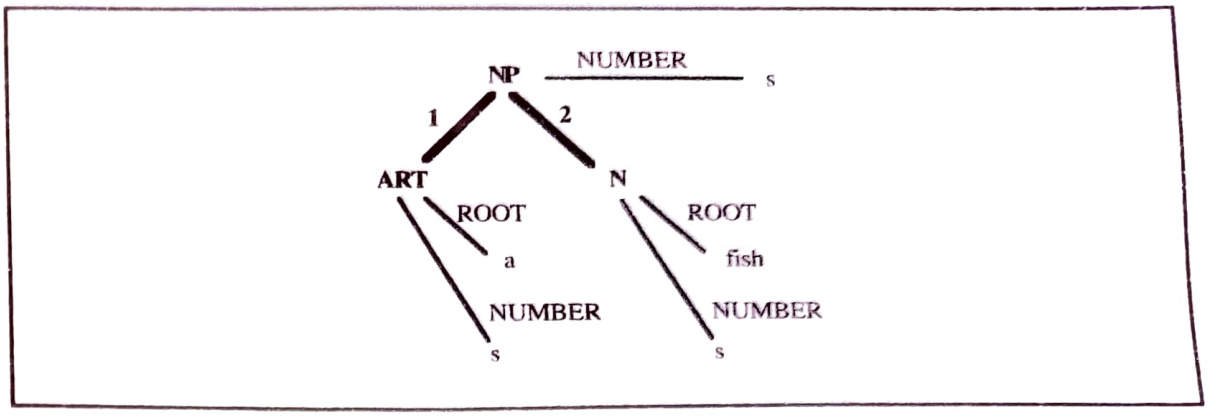


Figure 4.1 Viewing a feature structure as an extended parse tree

```
*(NP   1 (ART NUMBER s)
       2 (N NUMBER s))
```

is not allowed by this rule because there is no NUMBER feature in the NP, and the constituent

```
*(NP   NUMBER s
       1 (ART NUMBER s)
       2 (N NUMBER p))
```

is not allowed because the NUMBER feature of the N constituent is not identical to the other two NUMBER features.

Variables are also useful in specifying ambiguity in a constituent. For instance, the word *fish* is ambiguous between a singular and a plural reading. Thus the word might have two entries in the lexicon that differ only by the value of the NUMBER feature. Alternatively, we could define a single entry that uses a variable as the value of the NUMBER feature, that is,

```
(N ROOT fish NUMBER ?n)
```

This works because any value of the NUMBER feature is allowed for the word *fish*. In many cases, however, not just any value would work, but a range of values is possible. To handle these cases, we introduce **constrained variables**, which are variables that can only take a value out of a specified list. For example, the variable ?n{s p} would be a variable that can take the value *s* or the value *p*. Typically, when we write such variables, we will drop the variable name altogether and just list the possible values. Given this, the word *fish* might be represented by the constituent

```
(N ROOT fish NUMBER ?n{s p})
```

or more simply as

```
(N ROOT fish NUMBER {s p})
```

BOX 4.1 Formalizing Feature Structures

There is an active area of research in the formal properties of feature structures. This work views a feature system as a formal logic. A feature structure is defined as a partial function from features to feature values. For example, the feature structure

ART1: (**CAT** ART
ROOT a
NUMBER s)

is treated as an abbreviation of the following statement in FOPC:

$$ART1(CAT) = ART \wedge ART1(ROOT) = a \wedge ART1(NUMBER) = s$$

Feature structures with disjunctive values map to disjunctions. The structure

THE1: (**CAT** ART
ROOT the
NUMBER {s p})

would be represented as

$$THE1(CAT) = ART \wedge THE1(ROOT) = the$$
$$\wedge (THE1(NUMBER) = s \vee THE1(NUMBER) = p)$$

Given this, agreement between feature values can be defined as equality equations.

There is an interesting issue of whether an augmented context-free grammar can describe languages that cannot be described by a simple context-free grammar. The answer depends on the constraints on what can be a feature value. If the set of feature values is finite, then it would always be possible to create new constituent categories for every combination of features. Thus it is expressively equivalent to a context-free grammar. If the set of feature values is unconstrained, however, then such grammars have arbitrary computational power. In practice, even when the set of values is not explicitly restricted, this power is not used, and the standard parsing algorithms can be used on grammars that include features.

4.2 Some Basic Feature Systems for English

This section describes some basic feature systems that are commonly used in grammars of English and develops the particular set of features used throughout this book. Specifically, it considers number and person agreement, verb form features, and features required to handle subcategorization constraints. You should read this to become familiar with the features in general and then refer back to it later when you need a detailed specification of a particular feature.

Person and Number Features

In the previous section, you saw the number system in English: Words may be classified as to whether they can describe a single object or multiple objects. While number agreement restrictions occur in several different places in English, they are most importantly found in subject-verb agreement. But subjects and verbs must also agree on another dimension, namely with respect to the **person**. The possible values of this dimension are

First Person (1): The noun phrase refers to the speaker, or a group of people including the speaker (for example, *I, we, you, and I*).

Second Person (2): The noun phrase refers to the listener, or a group including the listener but not including the speaker (for example, *you, all of you*).

Third Person (3): The noun phrase refers to one or more objects, not including the speaker or hearer.

Since number and person features always co-occur, it is convenient to combine the two into a single feature, AGR, that has six possible values: first person singular (1s), second person singular (2s), third person singular (3s), and first, second and third person plural (1p, 2p, and 3p, respectively). For example, an instance of the word *is* can agree only with a third person singular subject, so its AGR feature would be 3s. An instance of the word *are*, however, may agree with second person singular or any of the plural forms, so its AGR feature would be a variable ranging over the values {2s 1p 2p 3p}.

Verb-Form Features and Verb Subcategorization

Another very important feature system in English involves the form of the verb. This feature is used in many situations, such as the analysis of auxiliaries and generally in the subcategorization restrictions of many head words. As described in Chapter 2, there are five basic forms of verbs. The feature system for verb forms will be slightly more complicated in order to conveniently capture certain phenomena. In particular, we will use the following feature values for the feature VFORM:

base—base form (for example, *go, be, say, decide*)

pres—simple present tense (for example, *go, goes, am, is, say, says, decide*)

past—simple past tense (for example, *went, was, said, decided*)

fin—finite (that is, a tensed form, equivalent to {pres past})

ing—present participle (for example, *going, being, saying, deciding*)

pastprt—past participle (for example, *gone, been, said, decided*)

inf—a special feature value that is used for infinitive forms with the word *to*

Value	Example Verb	Example
_none	laugh	Jack laughed.
_np	find	Jack found a key.
_np_np	give	Jack gave Sue the paper.
_vp:inf	want	Jack wants to fly.
_np_vp:inf	tell	Jack told the man to go.
_vp:ing	keep	Jack keeps hoping for the best.
_np_vp:ing	catch	Jack caught Sam looking at his desk.
_np_vp:base	watch	Jack watched Sam look at his desk.

Figure 4.2 The SUBCAT values for NP/VP combinations

To handle the interactions between words and their complements, an additional feature, SUBCAT, is used. Chapter 2 described some common verb subcategorization possibilities. Each one will correspond to a different value of the SUBCAT feature. Figure 4.2 shows some SUBCAT values for complements consisting of combinations of NPs and VPs. To help you remember the meaning of the feature values, they are formed by listing the main category of each part of the complement. If the category is restricted by a feature value, then the feature value follows the constituent separated by a colon. Thus the value _np_vp:inf will be used to indicate a complement that consists of an NP followed by a VP with VFORM value inf. Of course, this naming is just a convention to help the reader; you could give these values any arbitrary name, since their significance is determined solely by the grammar rules that involve the feature. For instance, the rule for verbs with a SUBCAT value of _np_vp:inf would be

(VP) → (V **SUBCAT** _np_vp:inf)
(NP)
(VP **VFORM** inf)

This says that a VP can consist of a V with SUBCAT value _np_vp:inf, followed by an NP, followed by a VP with VFORM value inf. Clearly, this rule could be rewritten using any other unique symbol instead of _np_vp:inf, as long as the lexicon is changed to use this new value.

Many verbs have complement structures that require a prepositional phrase with a particular preposition, or one that plays a particular role. For example, the verb *give* allows a complement consisting of an NP followed by a PP using the preposition *to*, as in *Jack gave the money to the bank.* Other verbs, such as *put,* require a prepositional phrase that describes a location, using prepositions such as *in, inside, on,* and *by.* To express this within the feature system, we introduce a feature PFORM on prepositional phrases. A prepositional phrase with a PFORM value such as TO must have the preposition *to* as its head, and so on. A prepositional phrase with a PFORM value LOC must describe a location. Another useful PFORM value is MOT, used with verbs such as *walk,* which may take a

Value	Example Prepositions	Example
TO	to	I gave it to the bank.
LOC	in, on, by, inside, on top of	I put it on the desk.
MOT	to, from, along, ...	I walked to the store.

Figure 4.3 Some values of the PFORM feature for prepositional phrases

Value	Example Verb	Example
_np_pp:to	give	Jack gave the key to the man.
_pp:loc	be	Jack is at the store.
_np_pp:loc	put	Jack put the box in the corner.
_pp:mot	go	Jack went to the store.
_np_pp:mot	take	Jack took the hat to the party.
_adjp	be, seem	Jack is happy.
_np_adjp	keep	Jack kept the dinner hot.
_s:that	believe	Jack believed that the world was flat.
_s:for	hope	Jack hoped for the man to win the prize.

Figure 4.4 Additional SUBCAT values

prepositional phrase that describes some aspect of a path, as in *We walked to the store.* Prepositions that can create such phrases include *to, from,* and *along.* The LOC and MOT values might seem hard to distinguish, as certain prepositions might describe either a location or a path, but they are distinct. For example, while *Jack put the box {in on by} the corner* is fine, **Jack put the box {to from along} the corner* is ill-formed. Figure 4.3 summarizes the PFORM feature.

This feature can be used to restrict the complement forms for various verbs. Using the naming convention discussed previously, the SUBCAT value of a verb such as *put* would be _np_pp:loc, and the appropriate rule in the grammar would be

(VP) → (V **SUBCAT** _np_pp:loc)
(NP)
(PP **PFORM** LOC)

For embedded sentences, a complementizer is often needed and must be subcategorized for. Thus a COMP feature with possible values *for, that,* and *no-comp* will be useful. For example, the verb *tell* can subcategorize for an S that has the complementizer *that.* Thus one SUBCAT value of *tell* will be _s:that. Similarly, the verb *wish* subcategorizes for an S with the complementizer *for,* as in *We wished for the rain to stop.* Thus one value of the SUBCAT feature for *wish* is _s:for. Figure 4.4 lists some of these additional SUBCAT values and examples for a variety of verbs. In this section, all the examples with the

SUBCAT feature have involved verbs, but nouns, prepositions, and adjectives may also use the SUBCAT feature and subcategorize for their complements in the same way.

Binary Features

Certain features are binary in that a constituent either has or doesn't have the feature. In our formalization a binary feature is simply a feature whose value is restricted to be either + or –. For example, the INV feature is a binary feature that indicates whether or not an S structure has an inverted subject (as in a yes/no question). The S structure for the sentence *Jack laughed* will have an INV value –, whereas the S structure for the sentence *Did Jack laugh?* will have the INV value +. Often, the value is used as a prefix, and we would say that a structure has the feature +INV or –INV. Other binary features will be introduced as necessary throughout the development of the grammars.

The Default Value for Features

It will be useful on many occasions to allow a **default value** for features. Anytime a constituent is constructed that could have a feature, but a value is not specified, the feature takes the default value of –. This is especially useful for binary features but is used for nonbinary features as well; this usually ensures that any later agreement check on the feature will fail. The default value is inserted when the constituent is first constructed.

4.3 Morphological Analysis and the Lexicon

Before you can specify a grammar, you must define the lexicon. This section explores some issues in lexicon design and the need for a morphological analysis component.

The lexicon must contain information about all the different words that can be used, including all the relevant feature value restrictions. When a word is ambiguous, it may be described by multiple entries in the lexicon, one for each different use.

Because words tend to follow regular morphological patterns, however, many forms of words need not be explicitly included in the lexicon. Most English verbs, for example, use the same set of suffixes to indicate different forms: *-s* is added for third person singular present tense, *-ed* for past tense, *-ing* for the present participle, and so on. Without any morphological analysis, the lexicon would have to contain every one of these forms. For the verb *want* this would require six entries, for *want* (both in base and present form), *wants, wanting,* and *wanted* (both in past and past participle form).

In contrast, by using the methods described in Section 3.7 to strip suffixes there needs to be only one entry for *want*. The idea is to store the base form of the

verb in the lexicon and use context-free rules to combine verbs with suffixes to derive the other entries. Consider the following rule for present tense verbs:

(V **ROOT** ?r **SUBCAT** ?s **VFORM** pres **AGR** 3s) →
(V **ROOT** ?r **SUBCAT** ?s **VFORM** base) (+S)

where +S is a new lexical category that contains only the suffix morpheme *-s*. This rule, coupled with the lexicon entry

want: (V **ROOT** want
SUBCAT {_np _vp:inf _np_vp:inf}
VFORM base)

would produce the following constituent given the input string *want -s*

want: (V **ROOT** want
SUBCAT {_np _vp:inf _np_vp:inf}
VFORM pres
AGR 3s)

Another rule would generate the constituents for the present tense form not in third person singular, which for most verbs is identical to the root form:

(V **ROOT** ?r **SUBCAT** ?s **VFORM** pres **AGR** {1s 2s 1p 2p 3p}) →
(V **ROOT** ?r **SUBCAT** ?s **VFORM** base)

But this rule needs to be modified in order to avoid generating erroneous interpretations. Currently, it can transform any base form verb into a present tense form, which is clearly wrong for some irregular verbs. For instance, the base form *be* cannot be used as a present form (for example, **We be at the store*). To cover these cases, a feature is introduced to identify irregular forms. Specifically, verbs with the binary feature +IRREG-PRES have irregular present tense forms. Now the rule above can be stated correctly:

(V **ROOT** ?r **SUBCAT** ?s **VFORM** pres **AGR** {1s 2s 1p 2p 3p}) →
(V **ROOT** ?r **SUBCAT** ?s **VFORM** base **IRREG-PRES** –)

Because of the default mechanism, the IRREG-PRES feature need only be specified on the irregular verbs. The regular verbs default to –, as desired. Similar binary features would be needed to flag irregular past forms (IRREG-PAST, such as *saw*), and to distinguish *-en* past participles from *-ed* past participles (EN-PASTPRT). These features restrict the application of the standard lexical rules, and the irregular forms are added explicitly to the lexicon. Grammar 4.5 gives a set of rules for deriving different verb and noun forms using these features.

Given a large set of features, the task of writing lexical entries appears very difficult. Most frameworks allow some mechanisms that help alleviate these problems. The first technique—allowing default values for features—has already been mentioned. With this capability, if an entry takes a default value for a given feature, then it need not be explicitly stated. Another commonly used technique is

Present Tense

1. (V **ROOT** ?r **SUBCAT** ?s **VFORM** pres **AGR** 3s) →
   (V **ROOT** ?r **SUBCAT** ?s **VFORM** base **IRREG-PRES** –) +S
2. (V **ROOT** ?r **SUBCAT** ?s **VFORM** pres **AGR** {1s 2s 1p 2p 3p}) →
   (V **ROOT** ?r **SUBCAT** ?s **VFORM** base **IRREG-PRES** –)

Past Tense

3. (V **ROOT** ?r **SUBCAT** ?s **VFORM** past **AGR** {1s 2s 3s 1p 2p 3p}) →
   (V **ROOT** ?r **SUBCAT** ?s **VFORM** base **IRREG-PAST** –) +ED

Past Participle

4. (V **ROOT** ?r **SUBCAT** ?s **VFORM** pastprt) →
   (V **ROOT** ?r **SUBCAT** ?s **VFORM** base **EN-PASTPRT** –) +ED
5. (V **ROOT** ?r **SUBCAT** ?s **VFORM** pastprt) →
   (V **ROOT** ?r **SUBCAT** ?s **VFORM** base **EN-PASTPRT** +) +EN

Present Participle

6. (V **ROOT** ?r **SUBCAT** ?s **VFORM** ing) →
   (V **ROOT** ?r **SUBCAT** ?s **VFORM** base) +ING

Plural Nouns

7. (N **ROOT** ?r **AGR** 3p) →
   (N **ROOT** ?r **AGR** 3s **IRREG-PL** –) +S

Grammar 4.5 Some lexical rules for common suffixes on verbs and nouns

to allow the lexicon writing to define clusters of features, and then indicate a cluster with a single symbol rather than listing them all. Later, additional techniques will be discussed that allow the inheritance of features in a feature hierarchy.

Figure 4.6 contains a small lexicon. It contains many of the words to be used in the examples that follow. It contains three entries for the word *saw*—as a noun, as a regular verb, and as the irregular past tense form of the verb *see*—as illustrated in the sentences

The saw was broken.
Jack wanted me to saw the board in half.
I saw Jack eat the pizza.

With an algorithm for stripping the suffixes and regularizing the spelling, as described in Section 3.7, the derived entries can be generated using any of the basic parsing algorithms on Grammar 4.5. With the lexicon in Figure 4.6 and Grammar 4.5, correct constituents for the following words can be derived: *been, being, cries, cried, crying, dogs, saws* (two interpretations), *sawed, sawing, seen, seeing, seeds, wants, wanting,* and *wanted.* For example, the word *cries* would be transformed into the sequence *cry +s*, and then rule 1 would produce the present tense entry from the base form in the lexicon.

```
a:      (CAT ART
         ROOT A1
         AGR 3s)
be:     (CAT V
         ROOT BE1
         VFORM base
         IRREG-PRES +
         IRREG-PAST +
         SUBCAT (_adjp _np))
cry:    (CAT V
         ROOT CRY1
         VFORM base
         SUBCAT _none)
dog:    (CAT N
         ROOT DOG1
         AGR 3s)
fish:   (CAT N
         ROOT FISH1
         AGR (3s 3p)
         IRREG-PL +)
happy:  (CAT ADJ
         SUBCAT _vp:inf)
he:     (CAT PRO
         ROOT HE1
         AGR 3s)
is:     (CAT V
         ROOT BE1
         VFORM pres
         SUBCAT (_adjp _np)
         AGR 3s)
Jack:   (CAT NAME
         AGR 3s)
man:    (CAT N1
         ROOT MAN1
         AGR 3s)
men:    (CAT N
         ROOT MAN1
         AGR 3p)
saw:    (CAT N
         ROOT SAW1
         AGR 3s)
saw:    (CAT V
         ROOT SAW2
         VFORM base
         SUBCAT _np)
saw:    (CAT V
         ROOT SEE1
         VFORM past
         SUBCAT _np)
see:    (CAT V
         ROOT SEE1
         VFORM base
         SUBCAT _np
         IRREG-PAST +
         EN-PASTPRT +)
seed:   (CAT N
         ROOT SEED1
         AGR 3s)
the:    (CAT ART
         ROOT THE1
         AGR (3s 3p))
to:     (CAT TO)
want:   (CAT V
         ROOT WANT1
         VFORM base
         SUBCAT (_np _vp:inf _np_vp:inf))
was:    (CAT V
         ROOT BE1
         VFORM past
         AGR (1s 3s)
         SUBCAT (_adjp _np))
were:   (CAT V
         ROOT BE
         VFORM past
         AGR (2s 1p 2p 3p)
         SUBCAT (_adjp _np))
```

Figure 4.6 A lexicon

Often a word will have multiple interpretations that use different entries and different lexical rules. The word *saws,* for instance, transformed into the sequence *saw +s,* can be a plural noun (via rule 7 and the first entry for *saw*), or the third person present form of the verb *saw* (via rule 1 and the second entry for *saw*). Note that rule 1 cannot apply to the third entry, as its VFORM is not base.

The success of this approach depends on being able to prohibit erroneous derivations, such as analyzing *seed* as the past tense of the verb *see*. This analysis will never be considered if the FST that strips suffixes is correctly designed. Specifically, the word *see* will not allow a transition to the states that allow the *-ed* suffix. But even if this were produced for some reason, the IRREG-PAST value + in the entry for *see* would prohibit rule 3 from applying.

4.4 A Simple Grammar Using Features

This section presents a simple grammar using the feature systems and lexicon developed in the earlier sections. It will handle sentences such as the following:

The man cries.
The men cry.
The man saw the dogs.
He wants the dog.
He wants to be happy.
He wants the man to see the dog.
He is happy to be a dog.

It does not find the following acceptable:

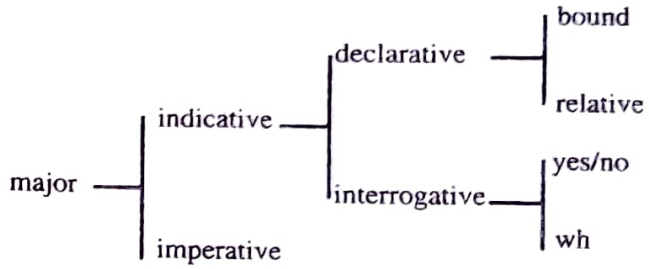
*The men cries.
*The man cry.
*The man saw to be happy.
*He wants.
*He wants the man saw the dog.

Before developing the grammar, some additional conventions are introduced that will be very useful throughout the book. It is very cumbersome to write grammatical rules that include all the necessary features. But there are certain regularities in the use of features that can be exploited to simplify the process of writing rules. For instance, many feature values are unique to a feature (for example, the value inf can only appear in the VFORM feature, and _np_vp:inf can only appear in the SUBCAT feature). Because of this, we can omit the feature name without introducing any ambiguity. Unique feature values will be listed using square parentheses. Thus (VP **SUBCAT** inf) will be abbreviated as VP[inf]. Since binary features do not have unique values, a special convention is introduced for them. For a binary feature B, the constituent C[+B] indicates the constituent (C **B** +).

Many features are constrained so that the value on the mother must be identical to the value on its head subconstituent. These are called **head features.** For instance, in all VP rules the VFORM and AGR values are the same in the VP and the head verb, as in the rule

BOX 4.2 Systemic Grammar

An important influence on the development of computational feature-based systems was **systemic grammar** (Halliday, 1985). This theory emphasizes the functional role of linguistic constructs as they affect communication. The grammar is organized as a set of choices about discourse function that determine the structure of the sentence. The choices are organized into hierarchical structures called **systems**. For example, the mood system would capture all the choices that affect the mood of the sentence. Part of this structure looks as follows:



This structure indicates that once certain choices are made, others become relevant. For instance, if you decide that a sentence is in the declarative mood, then the choice between bound and relative becomes relevant. The choice between yes/no and wh, on the other hand, is not relevant to a declarative sentence.

Systemic grammar was used in Winograd (1973), and Winograd (1983) contains a good discussion of the formalism. In recent years it has mainly been used in natural language generation systems because it provides a good formalism for organizing the choices that need to be made while planning a sentence (for example, see Mann and Mathiesson (1985) and Patten (1988)).

(VP **VFORM** ?v **AGR** ?a) →
(V **VFORM** ?v **AGR** ?a **SUBCAT** _np_vp:inf)
(NP)
(VP **VFORM** inf)

If the head features can be declared separately from the rules, the system can automatically add these features to the rules as needed. With VFORM and AGR declared as head features, the previous VP rule can be abbreviated as

VP → (*V* ***SUBCAT*** *_np_vp:inf*) NP (VP **VFORM** inf)

The head constituent in a rule will be indicated in italics. Combining all the abbreviation conventions, the rule could be further simplified to

VP → *V*[*_np_vp:inf*] NP VP[inf]

A simple grammar using these conventions is shown as Grammar 4.7. Except for rules 1 and 2, which must enforce number agreement, all the rest of the feature constraints can be captured using the conventions that have been

Consider why each of the ill-formed sentences introduced at the beginning of this section are not accepted by Grammar 4.7. Both **The men cries* and **The man cry* are not acceptable because the number agreement restriction on rule 1 is not satisfied: The NP constituent for *the men* has the AGR value 3p, while the VP *cries* has the AGR value 3s. Thus rule 1 cannot apply. Similarly, *the man cry* is not accepted by the grammar since *the man* has AGR 3s and the VP *cry* has as its AGR value a variable ranging over {1s 2s 1p 2p 3p}. The phrase **the man saw to be happy* is not accepted because the verb *saw* has a SUBCAT value _np. Thus only rule 5 could be used to build a VP. But rule 5 requires an NP complement, and it is not possible for the words *to be happy* to be a legal NP.

The phrase *He wants* is not accepted since the verb *wants* has a SUBCAT value ranging over {_np _vp:inf _np_vp:inf}, and thus only rules 5, 6, and 7 could apply to build a VP. But all these rules require a nonempty complement of some kind. The phrase **He wants the man saw the dog* is not accepted for similar reasons, but this requires a little more analysis. Again, rules 5, 6, and 7 are possible with the verb *wants*. Rules 5 and 6 will not work, but rule 7 looks close, as it requires an NP and a VP[inf]. The phrase *the man* gives us the required NP, but *saw the dog* fails to be a VP[inf]. In particular, *saw the man* is a legal VP, but its VFORM feature will be past, not inf.

4.5 Parsing with Features

The parsing algorithms developed in Chapter 3 for context-free grammars can be extended to handle augmented context-free grammars. This involves generalizing the algorithm for matching rules to constituents. For instance, the chart-parsing algorithms developed in Chapter 3 all used an operation for extending active arcs with a new constituent. A constituent X could extend an arc of the form

$$C \rightarrow C1 \ldots Ci \circ X \ldots Cn$$

to produce a new arc of the form

$$C \rightarrow C1 \ldots Ci\ X \circ \ldots Cn$$

A similar operation can be used for grammars with features, but the parser may have to instantiate variables in the original arc before it can be extended by X. The key to defining this matching operation precisely is to remember the definition of grammar rules with features. A rule such as

1. (NP **AGR** ?a) → ∘ (ART **AGR** ?a) (N **AGR** ?a)

says that an NP can be constructed out of an ART and an N if all three agree on the AGR feature. It does not place any restrictions on any other features that the NP, ART, or N may have. Thus, when matching constituents against this rule, the only thing that matters is the AGR feature. All other features in the constituent can be ignored. For instance, consider extending arc 1 with the constituent

2. (ART **ROOT** A **AGR** 3s)

To make arc 1 applicable, the variable ?a must be instantiated to 3s, producing

3. (NP **AGR** 3s) → ∘ (ART **AGR** 3s) (N **AGR** 3s)

This arc can now be extended because every feature in the rule is in constituent 2:

4. (NP **AGR** 3s) → (ART **AGR** 3s) ∘ (N **AGR** 3s)

Now, consider extending this arc with the constituent for the word *dog*:

5. (N **ROOT** DOG1 **AGR** 3s)

This can be done because the AGR features agree. This completes the arc

6. (NP **AGR** 3s) → (ART **AGR** 3s) (N **AGR** 3s) ∘

This means the parser has found a constituent of the form (NP **AGR** 3s).

This algorithm can be specified more precisely as follows: Given an arc A, where the constituent following the dot is called NEXT, and a new constituent X, which is being used to extend the arc,

a. Find an instantiation of the variables such that all the features specified in NEXT are found in X.
b. Create a new arc A', which is a copy of A except for the instantiations of the variables determined in step (a).
c. Update A' as usual in a chart parser.

For instance, let A be arc 1, and X be the ART constituent 2. Then NEXT will be (ART **AGR** ?a). In step a, NEXT is matched against X, and you find that ?a must be instantiated to 3s. In step b, a new copy of A is made, which is shown as arc 3. In step c, the arc is updated to produce the new arc shown as arc 4.

When constrained variables, such as ?a{3s 3p}, are involved, the matching proceeds in the same manner, but the variable binding must be one of the listed values. If a variable is used in a constituent, then one of its possible values must match the requirement in the rule. If both the rule and the constituent contain variables, the result is a variable ranging over the intersection of their allowed values. For instance, consider extending arc 1 with the constituent (ART **ROOT** the **AGR** ?v{3s 3p}), that is, the word *the*. To apply, the variable ?a would have to be instantiated to ?v{3s 3p}, producing the rule

(NP **AGR** ?v{3s 3p}) → (ART **AGR** ?v{3s 3p}) ∘ (N **AGR** ?v{3s 3p})

This arc could be extended by (N **ROOT** dog **AGR** 3s), because ?v{3s 3p} could be instantiated by the value 3s. The resulting arc would be identical to arc 6. The entry in the chart for *the* is not changed by this operation. It still has the value ?v{3s 3p}. The AGR feature is restricted to 3s only in the arc.

Another extension is useful for recording the structure of the parse. Subconstituent features (1, 2, and so on, depending on which subconstituent is being added) are automatically inserted by the parser each time an arc is extended. The values of these features name subconstituents already in the chart.

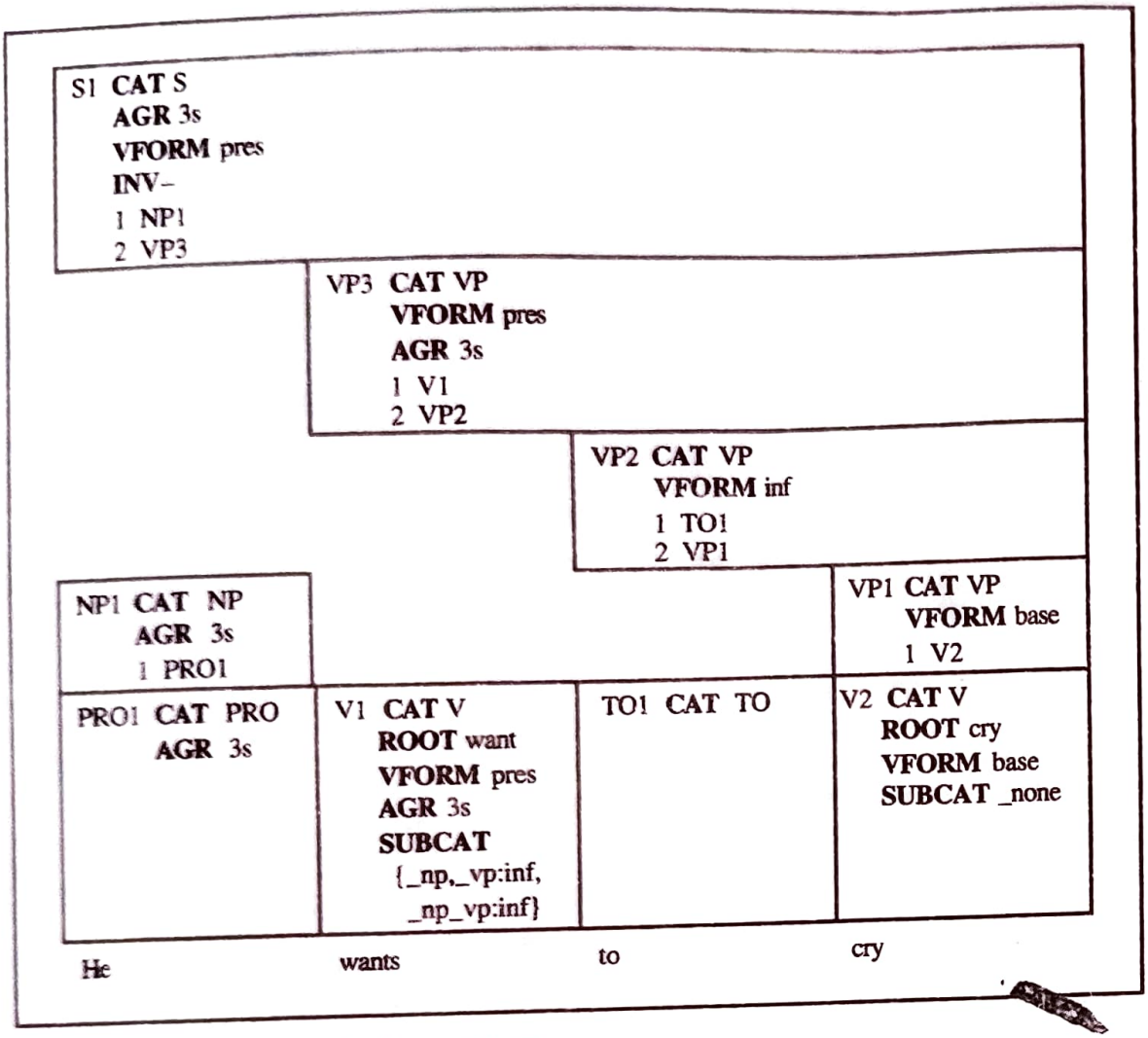


Figure 4.10 The chart for *He wants to cry*.

With this treatment, and assuming that the chart already contains two constituents, ART1 and N1, for the words *the* and *dog*, the constituent added to the chart for the phrase *the dog* would be

(NP **AGR** 3s
1 ART1
2 N1)

where ART1 = (ART **ROOT** the **AGR** {3s 3p}) and N1 = (N **ROOT** dog **AGR** {3s}). Note that the AGR feature of ART1 was not changed. Thus it could be used with other interpretations that require the value 3p if they are possible. Any of the chart-parsing algorithms described in Chapter 3 can now be used with an augmented grammar by using these extensions to extend arcs and build constituents. Consider an example. Figure 4.10 contains the final chart produced from parsing the sentence *He wants to cry* using Grammar 4.8. The rest of this section considers how some of the nonterminal symbols were constructed for the chart.

Constituent NP1 was constructed by rule 3, repeated here for convenience:

3. (NP **AGR** ?a) → (PRO **AGR** ?a)

To match the constituent PRO1, the variable ?a must be instantiated to 3s. Thus the new constituent built is

NP1: (**CAT** NP
AGR 3s
1 PRO1)

Next consider constructing constituent VP1 using rule 4, namely

4. (VP **AGR** ?a **VFORM** ?v) → (V **SUBCAT** _none **AGR** ?a **VFORM** ?v)

For the right-hand side to match constituent V2, the variable ?v must be instantiated to base. The AGR feature of V2 is not defined, so it defaults to –. The new constituent is

VP1: (**CAT** VP
AGR –
VFORM base
1 V2)

Generally, default values are not shown in the chart. In a similar way, constituent VP2 is built from TO1 and VP1 using rule 9, VP3 is built from V1 and VP2 using rule 6, and S1 is built from NP1 and VP3 using rule 1.

4.6 Augmented Transition Networks

Features can also be added to a recursive transition network to produce an **augmented transition network** (ATN). Features in an ATN are traditionally called **registers**. Constituent structures are created by allowing each network to have a set of registers. Each time a new network is pushed, a new set of registers is created. As the network is traversed, these registers are set to values by **actions** associated with each arc. When the network is popped, the registers are assembled to form a constituent structure, with the CAT slot being the network name.

Grammar 4.11 is a simple NP network. The actions are listed in the table below the network. ATNs use a special mechanism to extract the result of following an arc. When a lexical arc, such as arc 1, is followed, the constituent built from the word in the input is put into a special variable named *. The action

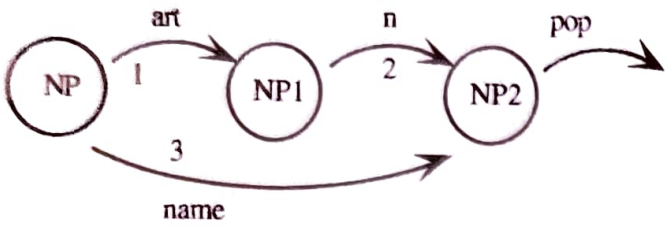
DET := *

then assigns this constituent to the DET register. The second action on this arc,

AGR := **AGR**$_*$

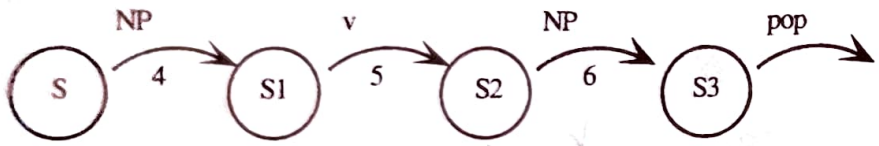
assigns the AGR register of the network to the value of the AGR register of the new word (the constituent in *).

Agreement checks are specified in the **tests**. A test is an expression that **succeeds** if it returns a nonempty value and **fails** if it returns the empty set or nil.



Arc	Test	Actions
1	none	**DET** := * **AGR** := **AGR**$_*$
2	**AGR** $\cap$ **AGR**$_*$	**HEAD** := * **AGR** := **AGR** $\cap$ **AGR**$_*$
3	none	**NAME** := * **AGR** := **AGR**$_*$

Grammar 4.11 A simple NP network



Arc	Test	Actions
4	none	**SUBJ** := *
5	**AGR**$_{\textbf{SUBJ}}$ $\cap$ **AGR**$_*$	**MAIN-V** := * **AGR** := **AGR**$_{\textbf{SUBJ}}$ $\cap$ **AGR**$_*$
6	none	**OBJ** := *

Grammar 4.12 A simple S network

If a test fails, its arc is not traversed. The test on arc 2 indicates that the arc can be followed only if the AGR feature of the network has a non-null intersection with the AGR register of the new word (the noun constituent in *).

Features on push arcs are treated similarly. The constituent built by traversing the NP network is returned as the value *. Thus in Grammar 4.12, the action on the arc from S to S1,

SUBJ := *

would assign the constituent returned by the NP network to the register SUBJ. The test on arc 2 will succeed only if the AGR register of the constituent in the SUBJ register has a non-null intersection with the AGR register of the new constituent (the verb). This test enforces subject-verb agreement.

Trace of S Network

Step	Node	Position	Arc Followed	Registers Set
1.	S	1	arc 4 succeeds (for recursive call see trace below)	**SUBJ** ← (NP **DET** the **HEAD** dog **AGR** 3s)
5.	S1	3	arc 5 (checks if 3p ∩ 3p)	**MAIN-V** ← saw **AGR** ← 3p
6.	S2	4	arc 6 (for recursive call trace, see below)	**OBJ** ← (NP **NAME** Jack **AGR** 3s)
9.	S3	5	pop arc succeeds	returns (S **SUBJ** (NP **DET** the **HEAD** dog **AGR** 3s) **MAIN-V** saw **AGR** 3p **OBJ** (NP **NAME** Jack **AGR** 3s))

Trace of First NP Call: Arc 4

Step	Node	Position	Arc Followed	Registers Set
2.	NP	1	1	**DET** ← the **AGR** ← {3s 3p}
3.	NP1	2	2 (checks if {3s 3p} ∩ 3p)	**HEAD** ← dog
4.	NP2	3	pop	returns (NP **DET** the **HEAD** dog **AGR** 3s)

Trace of Second NP Call: Arc 6

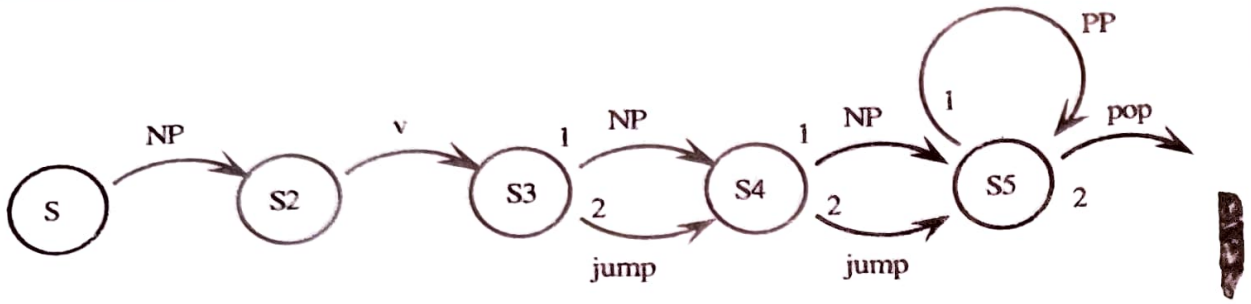
Step	Node	Position	Arc Followed	Registers Set
7.	NP	4	3	**NAME** ← John **AGR** ← 3s
8.	NP2	5	pop	returns (NP **NAME** John **AGR** 3s)

Figure 4.13 Trace tests and actions used with $_1$ *The* $_2$ *dog* $_3$ *saw* $_4$ *Jack* $_5$

With the lexicon in Section 4.3, the ATN accepts the following sentences:

The dog cried.
The dogs saw Jack.
Jack saw the dogs.

Consider an example. A trace of a parse of the sentence *The dog saw Jack* is shown in Figure 4.13. It indicates the current node in the network, the current



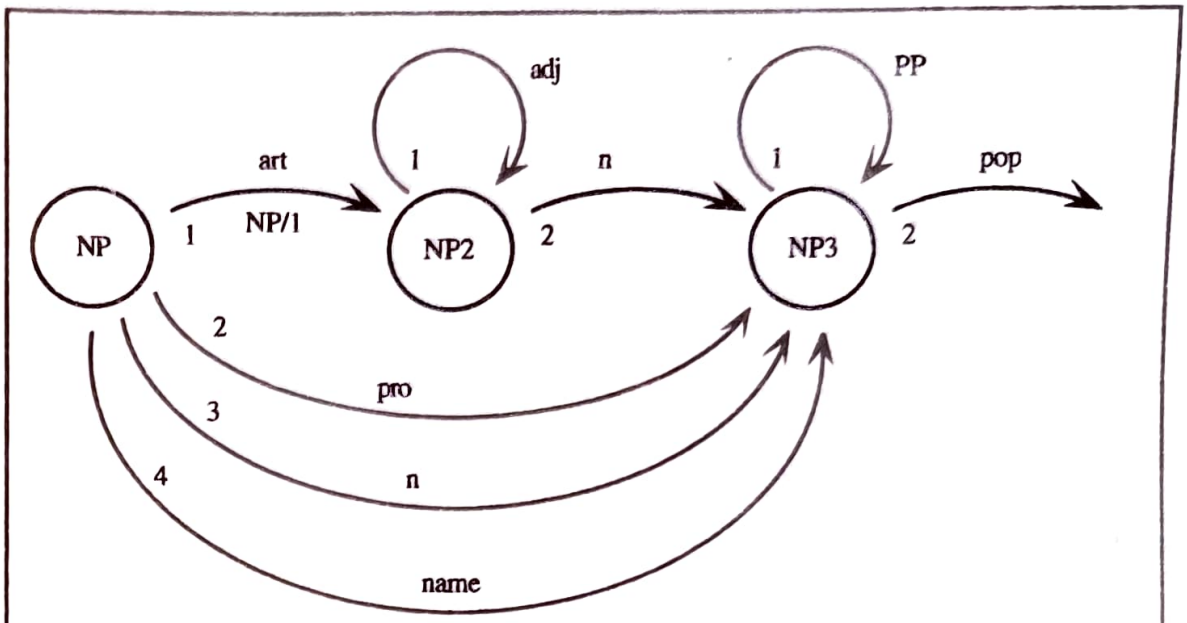
Arc	Test	Actions
S/1		**SUBJ** := *
		MOOD := DECL
S2/1	**AGR**$_{\text{SUBJ}}$ ∩ **AGR**$_{*}$	**MAIN-V** := *
S3/1	**SUBCAT**$_{\text{MAIN-V}}$ ∩ {_np _np_np}	**OBJ** := *
		SUBCAT := **SUBCAT**$_{\text{MAIN-V}}$ ∩ {_np _np_np}
S3/2	**SUBCAT**$_{\text{MAIN-V}}$ ∩ _none	**SUBCAT** := _none
S4/1	**SUBCAT**$_{\text{MAIN-V}}$ ∩ _np_np	**IOBJ** := **OBJ**
		OBJ := *
S5/1	---------	**MODS** := Append(**MODS**, *)

Grammar 4.14 An S network for assertions

word position, the arc that is followed from the node, and the register manipulations that are performed for the successful parse. It starts in the S network but moves immediately to the NP network from the call on arc 4. The NP network checks for number agreement as it accepts the word sequence *The dog*. It constructs a noun phrase with the **AGR** feature plural. When the pop arc is followed, it completes arc 4 in the S network. The NP is assigned to the **SUBJ** register and then checked for agreement with the verb when arc 3 is followed. The NP *Jack* is accepted in another call to the NP network.

An ATN Grammar for Simple Declarative Sentences

Here is a more comprehensive example of the use of an ATN to describe some declarative sentences. The allowed sentence structure is an initial NP followed by a main verb, which may then be followed by a maximum of two NPs and many PPs, depending on the verb. Using the feature system extensively, you can create a grammar that accepts any of the preceding complement forms, leaving the actual verb-complement agreement to the feature restrictions. Grammar 4.14 shows the S network. Arcs are numbered using the conventions discussed in Chapter 3. For instance, the arc S3/1 is the arc labeled 1 leaving node S3. The NP network in Grammar 4.15 allows simple names, bare plural nouns, pronouns, and a simple sequence of a determiner followed by an adjective and a head noun.



Arc	Test	Actions
NP/1	---------	**DET** := *
NP/2	---------	**PRO** := *
NP/3	**AGR**$_*$ ∩ 3p	**HEAD** := * **AGR** := **AGR**$_*$
NP/4	---------	**NAME** := *
NP2/1	---------	**ADJS** := Append(**ADJS**, *)
NP2/2	**AGR** ∩ **AGR**$_*$	**HEAD** := * **AGR** := **AGR** ∩ **AGR**$_*$
NP3/1	---------	**MODS** := Append(**MODS**, *)

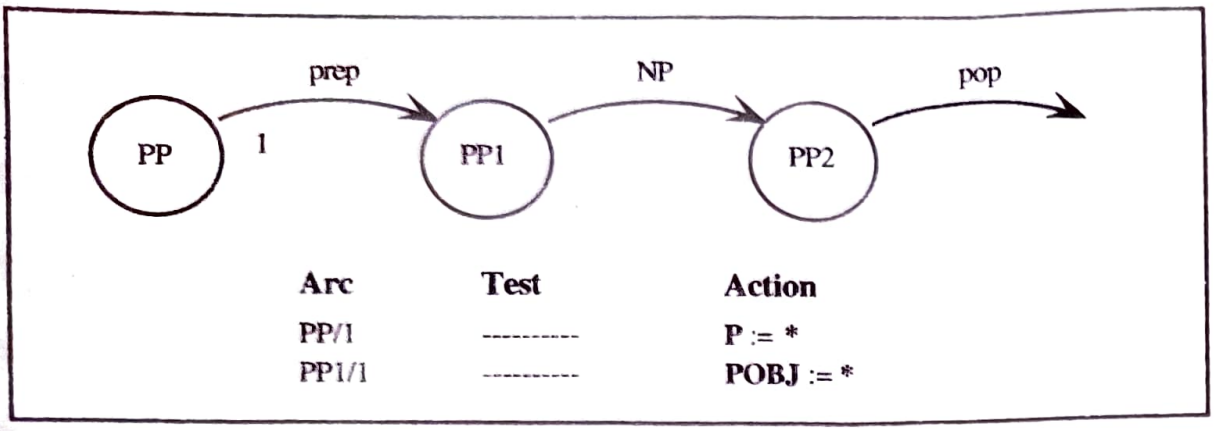
Grammar 4.15 The NP network

Allowable noun complements include an optional number of prepositional phrases. The prepositional phrase network in Grammar 4.16 is straightforward. Examples of parsing sentences with this grammar are left for the exercises.

Presetting Registers

One further extension to the feature-manipulation facilities in ATNs involves the ability to preset registers in a network as that network is being called, much like parameter passing in a programming language. This facility, called the **SENDR** action in the original ATN systems, is useful to pass information to the network that aids in analyzing the new constituent.

Consider the class of verbs, including *want* and *pray*, that accept complements using the infinitive forms of verbs, which are introduced by the word *to*. According to the classification in Section 4.2, this includes the following:



Grammar 4.16 The PP network

_vp:inf	Mary wants *to have a party.*
_np_vp:inf	Mary wants John *to have a party.*

In the context-free grammar developed earlier, such complements were treated as VPs with the VFORM value inf. To capture this same analysis in an ATN, you would need to be able to call a network corresponding to VPs but preset the VFORM register in that network to inf. Another common analysis of these constructs is to view the complements as a special form of sentence with an understood subject. In the first case it is Mary who would be the understood subject (that is, the host), while in the other case it is John. To capture this analysis, many ATN grammars preset the SUBJ register in the new S network when it is called.

4.7 Definite Clause Grammars

You can augment a logic grammar by adding extra arguments to each predicate to encode features. As a very simple example, you could modify the PROLOG rules to enforce number agreement by adding an extra argument for the number on every predicate for which the number feature is relevant. Thus you would have rules such as those shown in Grammar 4.17.

Consider parsing the noun phrase *the dog cried,* which would be captured by the assertions

```
word(the, 1, 2) :-
word(dog, 2, 3) :-
```

With these axioms, when the word *the* is parsed by rule 2 in Grammar 4.17, the number feature for the word is returned. You can see this in the following trace of the simple proof of

```
np(1; Number, 3)
```

Using rule 1, you have the following subgoals:

```
1.  np(P1, Number, P3) :– art(P1, Number, P2), n(2, Number, P3)
2.  art(I, Number, O) :– word(Word, I, O), isart(Word, Number)
3.  isart(a, 3s) :–
4.  isart(the, 3s) :–
5.  isart(the, 3p) :–
6.  n(I, Number, O) :– word(Word, I, O), isnoun(Word, Number)
7.  isnoun(dog, 3s) :–
8.  isnoun(dogs, 3p) :–
```

Grammar 4.17

```
1.  s(P1, Number, s(Np, Vp), P3) :–
        np(P1, Number, Np, P2), vp(P2, Number, Vp, P3)
2.  np(P1, Number, np(Art, N), P3) :–
        art(P1, Number1, Art, P2), n(P2, Number2, N, P3)
3.  vp(P1, Number, vp(Verb), P2) :–
        v(P1, Verb, P2)
4.  art(I, Number, art(Word), O) :–
        word(Word, I, O), isart(Word, Number)
5.  n(I, Number, n(Word), O) :–
        word(Word, I, O), isnoun(Word, Number)
6.  v(I, Number, v(Word), O) :–
        word(Word, I, O), isverb(Word, Number)
```

Grammar 4.18

art(1, Number, P2)
n(P2, Number, 3)

The first subgoal succeeds by using rule 2 and proving

word(the, 1, 2) isart(the, 3s)

which binds the variables in rule 2 as follows:

Number ← 3s
P2 ← 2

Thus, the second subgoal now is

n(2, 3s, 3)

Using rule 6, this reduces to the subgoals word(Word, 2, 3) and isnoun(Word, 3s), which are established by the input and rule 7, respectively, with Word bound to dog. Thus the parse succeeds and the number agreement was enforced.

The grammar can also be extended to record the structure of the parse by adding another argument to the rules. For example, to construct a parse tree, you could use the rules shown in Grammar 4.18. These rules would allow you to prove the following on the sentence *The dog cried*:

the feature type and the value simultaneously. For example, a plural noun phrase in the third person with gender female would be of the grammatical type

NP[PL,3,F]

Since the number of feature values is finite, the grammar is formally equivalent to a context-free grammar with a symbol for every combination of categories and features. One of the important contributions of GPSG, however, is the rich structure it imposes on the propagation of features. Rather than using explicit feature equation rules, GPSG relies on general principles of feature propagation that apply to all rules in the grammar. A good example of such a general principle is the **head feature convention**, which states that all the head features on the parent constituent must be identical to its head constituent. Another general principle enforces agreement restrictions between constituents. Some of the conventions introduced in this chapter to reduce the number of feature equations that must be defined by hand for each rule are motivated by these theoretical claims. An excellent survey of GPSG and LFG is found in Sells (1985).

The ATN framework described here is drawn from the work described in Woods (1970; 1973) and Kaplan (1973). A good survey of ATNs is Bates (1978). Logic programming approaches to natural language parsing originated in the early 1970s with work by Colmerauer (1978). This approach is perhaps best described in a paper by Pereira and Warren (1980) and in Pereira and Shieber (1987). Other interesting developments can be found in McCord (1980) and Pereira (1981). A good recent example is Alshawi (1992).

The discussion of unification grammars is based loosely on work by Kay (1982) and the PATR-II system (Shieber, 1984; 1986). There is a considerable amount of active research in this area. An excellent survey of the area is found in Shieber (1986). There is also a growing body of work on formalizing different forms of feature structures. Good examples are Rounds (1988), Shieber (1992), and Johnson (1991).

Exercises for Chapter 4

1. (*easy*) Using Grammar 4.7, draw the complete charts resulting from parsing the two sentences *The man cries* and *He wants to be happy,* whose final analyses are shown in Figure 4.9. You may use any parsing algorithm you want, but make sure that you state which one you are using and that the final chart contains every completed constituent built during the parse.

2. (*medium*) Define the minimal set of lexicon entries for the following verbs so that, using the morphological analysis algorithm, all standard forms of the verb are recognized and no illegal forms are inadvertently produced. Discuss any problems that arise and assumptions you make, and suggest modifications to the algorithms presented here if needed.

Base	Present Forms	Past	Past-Participle	Present-Participle
go	go, goes	went	gone	going
sing	sing, sings	sang	sung	singing
bid	bid, bids	bid	bidden	bidding

3. (*medium*) Extend the lexicon in Figure 4.6 and Grammar 4.7 so that the following two sentences are accepted:

He was sad to see the dog cry.
He saw the man saw the wood with the saw.

Justify your new rules by showing that they correctly handle a range of similar cases. Either implement and test your extended grammar using the supplied parser, or draw out the full chart that would be constructed for each sentence by a top-down chart parser.

4. (*medium*)

a. Write a grammar with features that will successfully allow the following phrases as noun phrases:

three o'clock
ten minutes to six
half past four
quarter after eight
seven thirty-five

but will not permit the following:

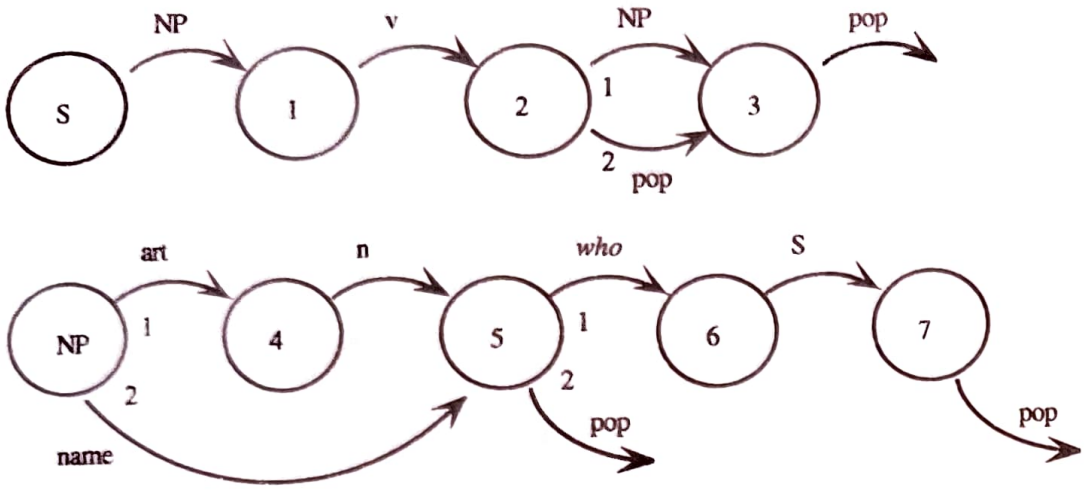
half to eight
ten forty-five after six
three twenty o'clock

Specify the feature manipulations necessary so that once the parse is completed, two features, HOUR and MINUTES, are set in the NP constituent. If this requires an extension to the feature mechanism, carefully describe the extension you assume.

b. Choose two other forms of grammatical phrases accepted by the grammar. Find an acceptable phrase not accepted by your grammar. If any nongrammatical phrases are allowed, give one example.

5. (*medium*) English pronouns distinguish case. Thus *I* can be used as a subject, and *me* can be used as an object. Similarly, there is a difference between *he* and *him*, *we* and *us*, and *they* and *them*. The distinction is not made for the pronoun *you*. Specify an augmented context-free grammar and lexicon for simple subject-verb-object sentences that allows only appropriate pronouns in the subject and object positions and does number agreement between the subject and verb. Thus it should accept *I hit him*, but not *me love you*. Your grammar should account for all the pronouns mentioned in this question, but it need have only one verb entry and need cover no other noun phrases but pronouns.

6. (*medium*) Consider the following simple ATN:



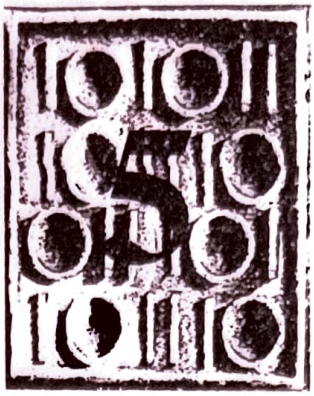
a. Specify some words by category and give four structurally different sentences accepted by this network.

b. Specify an augmentation for this network in the notation defined in this chapter so that sentences with the main verb *give* are allowed only if the subject is animate, and sentences with the main verb *be* may take either an animate or inanimate subject. Show a lexicon containing a few words that can be used to demonstrate the network's selectivity.

7. (*medium*) Using the following unification grammar, draw the DAGs for the two NP structures as they are when they are first constructed by the parser, and then give the DAG for the complete sentence (which will include all subconstituents of S as well) *The fish is a large one.* You may assume the lexicon in Figure 4.6, but define lexical entries for any words not covered there.

1.	S → NP VP	**INV** = – **VFORM**$_2$ = pres **AGR** = **AGR**$_1$ = **AGR**$_2$
2.	NP → ART N	**AGR** = **AGR**$_1$ = **AGR**$_2$
3.	NP → ART ADJ N	**AGR** = **AGR**$_1$ = **AGR**$_3$
4.	VP → V NP	**VFORM** = **VFORM**$_1$ **AGR** = **AGR**$_1$ = **AGR**$_2$ **ROOT** = BE1

CHAPTER



Grammars for Natural Language