Language as a Cognitive Process

Volume I: Syntax

Terry Winograd
Stanford University

Addison-Wesley Publishing Company
Reading, Massachusetts · Menlo Park, California · London
Amsterdam · Don Mills, Ontario · Sydney

Chapter 2

Word Patterns and Word Classes

Syntax is the part of linguistics that deals with how the words of a language are arranged into phrases and sentences and how components (like prefixes and suffixes) are combined to make words. In theory, it would not be necessary for languages to have a systematic syntax. We could imagine, for example, a language that was simply a list of all the things that could be said. The linguist's work would consist of compiling giant dictionaries of all the possible phrases with the meaning of each. In fact, there are *finite languages* for which such a dictionary exists, such as those of military and diplomatic code books. Even in ordinary conversation, many of our utterances are copied whole from a stock of phrases and cliches, including social formulas, such as *How do you do?,* and expressions, such as *The more the merrier* and *It takes one to know one.*

Human language taken as a whole, though, is infinite. We can produce sentences that we have never heard or spoken before, and they can be understood by others for whom they are totally new. At the other extreme from a finite language we could imagine a completely free language in which any sequence of words that had a possible interpretation was in the language. The sentence *Language interesting is* would be just as reasonable as *Language is interesting,* since it would have a clear interpretation. But no human language is syntax-free. Our freedom to create novel utterances operates within a framework of *grammar,* which puts strong constraints on the *patterns* that are used in the language.

Chapters 2 through 6 present a view of the knowledge needed by a language user to interpret and produce syntactic structures, and some mechanisms are given by which the processing can be accomplished. In this chapter, we will look at some simple kinds of linguistic patterns and introduce some of the computational mechanisms that will be used throughout the book. Section 2.1 describes the idea of patterns and pattern matching in an elementary form and introduces the notation for describing objects and procedures. Section 2.2 describes the classification of words and its use in matching. Section 2.3 describes a more complex kind of pattern represented in a *transition network,* and Section 2.4 gives some procedures for recognizing sentences of a language using such networks.

2.1 Patterns and matching

The notion of *pattern* at its simplest is that of a physical object whose form is identical to the form of a piece of material to be cut. It can be used to determine the shape of an infinite variety of garments, differing in material, color, and texture. A comparable idea of linguistic patterns can be used to describe the possible forms of a language. Individual sentences such as *Traveling is a pleasure* can be viewed as being 'cut out' on the basis of more general forms that have blanks in place of specific words, such as '_ *is a* _.'

Some of the early computer programs that interacted with people in English used these simple patterns. Figure 2–1 lists the entire set of patterns used by SIR (Raphael, 1967). Of course, it was clear that this was an extremely limited part of English, and the importance of the program lay not in its handling of

_ is _.	Is _ _?
_ has _.	How many _ does _ have?
_ owns _.	How many _ does _ own?
Where is _?	What is the _ of _?
Is _ part of _?	How many _ are parts of _?
Does _ own _?	How many _ are there on _?
_ is _ part of _.	_ has as a part one _.
There are _ on _.	There is one _ on _.
_ is to the left of _.	_ is just to the left of _.
Is _ to the left of _?	Is _ just to the left of _?
_ is to the right of _.	_ is just to the right of _.
Is _ to the right of _?	Is _ just to the right of _?

Figure 2–1. Patterns recognized by SIR.

Search for a match in a set of patterns	
Purpose: Test whether a sequence of words matches any pattern	
Inputs: a sequence of words and a set of *patterns*	2-3
Basic Method: For each pattern in the set:	
If the *pattern matches the sequence of words*, succeed.	2-3
Conditions:	
If every element of the set is tested without a match, fail.	

Figure 2–2. Search for a match in a set of patterns.

syntax, but in the reasoning mechanisms it used to answer questions (which will be described in the volume on meaning).

The knowledge of syntax represented in such a program consists of a set of alternative sentence patterns, each specifying a particular sequence of words and places for words. A sequence of words is a sentence of the language if there is some pattern in the set which matches it. The patterns are used independently—a single pattern matches a whole sentence. In later chapters we will see more complex uses of patterns in which a sentence is described in terms of several patterns applying jointly. For the simple mechanisms of this chapter, we will deal only with sentences that can be matched by a single pattern.

A pattern matching procedure

As an introduction to the notation used for describing procedures and knowledge structures in this book, we will explain the definition of a simple pattern matcher in detail. The mechanism used here may seem overly complex for the structures being described, since it is being introduced in a very simple case to make clear just what the notations mean and how they are used.

Figure 2–2 describes how a set of patterns like those in Figure 2–1 could be used in a recognition procedure. The procedure goes through the patterns one at a time, stopping as soon as it finds one that fits. The input to the procedure is a word sequence, and successful recognition of the sequence means that it is a sentence of the language characterized by the set of patterns. We have not described here just what a 'word' is, but the definition will be discussed later in the chapter. In a full language understander, the input would be a sequence of sounds or written characters, and some other part of the language analysis process would divide it into words.

The definition is written in DL, a notation developed for this book and explained in Appendix A. Each definition describes a *procedure* (as this one does), a *class* of objects (as in the definition of 'pattern' in Figure 2–3), or a *predicate* used in logical expressions. The numbers to the right of the box are the figure numbers of definitions for classes of objects, procedures, and predicates

that are used in this definition. In each case, the term being cross-referenced appears in italics somewhere in the line next to which the number appears. A cross-reference is given only for the first appearance of a term in a particular figure, and will not be given for terms related to standard entities (such as words, characters, and sequences), which are used throughout the book and defined in Section A.4.

Several features of the definition deserve note:

Undefined objects, steps, and expressions. In describing this basic matching procedure, we have not said just what a pattern is or what it means for a pattern to match a word sequence. Any one of a number of different definitions for pattern could be 'plugged in' and the procedure would work in the same way. A general feature of descriptions in DL (and programming languages in general) is that we can write definitions that make use of objects, predicates, or procedures that are defined independently. If we look at this definition alone, it gives us an outline of what is to be done, but it is not detailed enough to actually carry it out.

The ultimate goal in designing a procedure is to make it complete and precise enough to be carried out by an *interpreter,* either a person or a program, which has the basic ability to carry out a collection of primitive steps. Appendix A describes the primitives of the DL interpreter. They include primitive objects such as sets and characters, primitive procedures such as stepping through a sequence, and primitive predicates such as equality. A procedure definition is a *fully defined algorithm* if each step, object, or expression is either a primitive of the language or refers to a definition that in turn is fully defined. We will discuss later what it takes for an object or predicate to be fully defined. Careful readers will note that this description of what it means for a procedure to be fully defined does not deal with *recursive definitions*—those that include a step making use of the definition in which it appears. For the moment, no such problems arise. See Section A.3 for a more comprehensive discussion.

Unspecified order. In saying 'For each pattern in the set...' we have not specified in what order to take them. For our purposes in this definition it does not make any difference, as long as they are taken one by one until a match succeeds or they have all been tried. One of the features of DL is that we can avoid being specific about ordering when it is irrelevant. A definition that includes a series of steps with an unspecified order is considered a fully defined algorithm, since any interpreter that actually carried out these instructions could choose some order arbitrarily. Of course, there are times when we want to be more specific. For example, the pattern set might contain two patterns that could apply to the same sequence of words, such as 'X _' and '_ Y', which both apply to 'X Y'. The procedure as we have described it would find one or the other but does not determine which. If the procedure used an ordered *sequence* of patterns instead of a set, we could determine which one would be found by the order in which they appeared.

PATTERN

Describes: a pattern to match a sequence of words

Kinds of Pattern: *Simple, Variable* 2–4 2–6

Basic Structure: a sequence, the class of each member depending on the kind of pattern

Predicates:

A pattern matches a sequence of words

Depends on the kind of pattern

Figure 2–3. Pattern.

Success and failure. The description of what to do for each pattern indicates that if it matches, the search will succeed. Once a successful pattern is found, no more are tried, even though the instruction says 'For each pattern....' Similarly, at the end, if nothing has been found the procedure fails. A procedure can include any number of steps that call for it to 'Succeed' or 'Fail.' Whenever such a step is reached in following the procedure, it has two consequences—the immediate stopping of the procedure and the determination of its outcome as success or failure.

Results. In many procedure descriptions, we want to describe some *results* that are produced. In Figure 2–2 we have not—the only result is that the search procedure succeeds or fails. It is an example of a program for *recognition* rather than for *parsing* or *understanding.* In most real applications, we are not interested in simply recognizing the fact that a sequence of words is a sentence of a language. We want to determine its structure and use it in some other procedure, such as question answering. A parsing procedure has as its result a structure describing the organization of the sequence of words as a sentence, while an understanding procedure produces an interpretation based on some notion of meaning. In most of this volume, we will be dealing with parsing—producing structures that are not interpreted for meaning, but which show the internal organization of the sentence. However, it is often useful to explain parsing procedures by first explaining the corresponding recognition procedures and then adding the additional detail needed to produce a structure.

A formal definition of patterns

Figure 2–3 gives a formal definition of patterns that begins to fill in some of the detail missing from Figure 2–2. It is still quite general, describing what is common to all patterns and indicating two specific kinds—*simple patterns,* in which

the elements are matched independently, and *variable patterns,* in which variables are used to keep track of what was matched for each element (explained below).

This definition illustrates three additional features of the notation:

Classes and kinds. The definition describes objects of the class 'Pattern.' Class definitions are indicated by a label running down the left side. Figure 2–3 indicates that there are two specific kinds (or *subclasses*) of patterns, each having its own definition (as indicated by the cross-reference numbers on the right). Those properties common to all kinds of patterns are included in this definition, while those specific to one kind appear in its definition.

Basic structure. Each pattern is in turn made up of a sequence of pattern elements. Not every kind of object has such a simple structure. For example, in Chapter 3 we will define a 'phrase structure node' as having 'roles' consisting of a 'label,' a 'parent,' and a set of 'children.' In the case of simple structures like the one defined in Figure 2–3, it is sufficient to indicate that it is a set or sequence and to say what class the elements belong to.

Predicates. A definition of a predicate such as 'A pattern matches a word sequence' is different from a procedure definition in that it describes the logical conditions for something to be true rather than a procedure to be carried out. There are primitive predicates in the language, such as equality of two objects and membership of an object in a set. These can be combined using logical operators such as 'not,' 'and,' and 'or.' It is also possible to define a predicate by giving the definition of a procedure that tests whether it is true or not, as is done in Figure 2–4. A predicate is *fully defined* if: it is primitive; or there is a fully defined algorithm for testing it; or it is defined as a combination of logical operators and fully defined predicates. Predicate definitions are indicated by underlining the phrase for the predicate. In Figure 2–4 we do not actually give the definition, leaving it to be defined for each kind of pattern. However, it is included here inside the definition of pattern to indicate that for every kind of pattern such a predicate must be provided.

Figure 2–4 gives yet more detail, providing a procedure by which we can test whether a pattern matches a sequence of words. This procedure is the obvious one of running through the pattern and sequence in parallel (a primitive procedure of the DL interpreter), checking to see if the elements match. However, the question of what it means for an element to match is once again left open to allow for different kinds of elements in patterns. Other things to note are:

Class hierarchy. A simple pattern is a kind of pattern, and in turn there are three kinds of simple patterns. We can describe a *hierarchy* of this sort to any depth. Anything appearing in a definition applies to all of the subclasses to any depth. A literal simple pattern is a kind of simple pattern and is therefore also a kind of pattern. Everything appearing in the definition of pattern (Figure 2–3) applies to it as well.

SIMPLE PATTERN

Describes: a pattern that matches a sequence of words with each element independent of the others

A Kind of: *Pattern* 2–3

Kinds of Simple Pattern: *Literal, Open, Lexical* 2–5 2–9

Basic Structure: a sequence, the class of each member depending on the kind of pattern

Procedures:

Test whether a pattern matches a sequence of words

Inputs: a pattern and a sequence of words

Basic Method: Step through the pattern and the sequence of words in parallel doing:

- If *the element of the pattern matches the word,* then go on. Otherwise fail. §

Conditions:

- If either sequence runs out before the other, fail.
- If both sequences run out simultaneously, succeed.

Predicates:

An element of a pattern matches a word

Depends on the kind of pattern

Figure 2–4. Simple pattern.

Nested definitions. The box containing the definition of simple pattern has within it a box defining the procedure 'Test whether a pattern matches a sequence of words.' This could have appeared in a separate figure, since it is a full definition. However, by including it inside the definition of this kind of pattern, we indicate that it constitutes a basic part of our understanding of what a simple pattern is. Without some notion of what it means to match a pattern, its definition as a sequence of elements would be uninteresting. In carrying out the procedure defined in Figure 2–2, the interpreter needs to use the appropriate definition of matching for the particular kind of pattern. In general, we will include definitions inside other definitions to indicate this kind of relevance. The character '§' is used in place of a cross-reference number when the definition being referred to appears in the same figure.

Procedures for testing predicates. The procedure defined within Figure 2–4 is the means of testing whether a pattern matches. It corresponds to the predicate that was mentioned in Figure 2–3. If the procedure succeeds for a given pattern and sequence, then it is true that the pattern matches the sequence. If it fails, the corresponding expression is false.

LITERAL PATTERN 2–4

Describes: a pattern with every element specified

A Kind of: *Simple pattern*

Basic Structure: a sequence of words

Predicates:

An element of a pattern matches a word
if it is equal to the word.

OPEN PATTERN 2–4

Describes: A pattern including a 'wildcard' that matches anything

A Kind of: *Simple pattern*

Basic Structure: a sequence, each member of which is either a word or the character '␣'

The choice of '␣' is arbitrary. All that matters is that there be a recognizable symbol that is allowed to match any word.

Predicates:

An element of a pattern matches a word
if it is equal to the word or is the character '␣'.

Figure 2–5. Two kinds of simple patterns.

Conditions. The procedure for matching a pattern includes conditions indicating what is to happen when the sequences run out. One feature of DL that is different from many programming languages is the ability to separate out special conditions like these from the description of the basic method. Whenever a condition is true of the current state of things, whatever it says to do is done, which may involve the success or failure of the procedure as a whole. The details of what can be included in such conditions are given in Section A.4.

Figure 2–5 defines two kinds of patterns, a literal pattern (a sequence to be matched exactly) and an open pattern of the kind shown for SIR in Figure 2–1. It fills in more details that were left open by the definition of simple patterns in Figure 2–4, specifying what the pattern elements are and what it means for an element to match a word. Comments appearing in italics are not part of the formal definition but are included as explanation. With this definition, we have a fully defined algorithm for searching for a match in a set of literal or open patterns, since all of the objects, steps, and expressions for which we have not given definitions are primitive. Note that the predicate 'An element of a pattern matches a word' is defined not by giving a test procedure, but by a logical expression built up out of primitive tests for equality and the logical 'or.'

VARIABLE PATTERN 2–3

Describes: a pattern whose elements are words and variables

A Kind of: *Pattern*

Basic Structure: a sequence, each member of which is either a word or an integer

Integers indicate the variables. If the same integer appears more than once, it must match the same word in all occurrences.

Procedures:

Match a variable pattern against a sequence of words

Purpose: Produce a table associating variables with words in the sequence

Inputs: a pattern and a sequence of words

Working Structures:

Bindings: a table whose keys are integers and whose entries are words; initially empty

Results: The table of bindings when the match is done

Basic Method: Step through the pattern and the sequence of words in parallel doing:

If the pattern element is:

- a word, then:
  - If it is the same as the corresponding word in the sequence go on. Otherwise the match fails.
- an integer, then:
  - If there is an entry for that integer in the bindings, then:
    - If the word in the sequence is equal to the entry, go on. Otherwise the match fails.
  - If there is no entry, add the word as an entry in the table with the integer as the key and go on.

Conditions:

If either sequence runs out before the other, the match fails.

If both sequences run out simultaneously then return the bindings.

Figure 2–6. Variable pattern.

Patterns with variables

The matching procedure of Figure 2–2 applied to simple patterns as defined in Figure 2–4 would not be very useful for a real language analyzer. Once it has finished its work, all we know is whether it succeeded or failed. There is no trace left of what words were matched against the pattern elements, or even of which pattern the whole sequence matched.

In order to perform a task like question answering, the input analyzer must not only see that the input is a real sentence, but it must also gather information

Pattern	Word	Bindings	Word	Bindings
1	*row*	1=*row*	*please*	1=*please*
2	*row*	1=*row*, 2=*row*	*turn*	1=*please*, 2=*turn*
2	*row*	1=*row*, 2=*row*	*in*	FAIL
your	*your*	1=*row*, 2=*row*	*your*	
3	*boat*	1=*row*, 2=*row*, 3=*boat*	*exam*	

Figure 2-7. Matching a pattern with variables.

on what was in it. The simplest mechanism for gathering information is to let the blanks be associated with *variables,* and to keep a pairing of these variables with the words that they matched. SIR in fact used variables of this kind.

Figure 2–6 defines a pattern with variables and its use in a more complex matching procedure. To distinguish variables from English words in patterns, we use integers. Figure 2–7 illustrates the sequence of steps in matching the pattern '1 2 2 your 3' against the sequences *row row row your boat* and *please turn in your exam.*

The definition of Figure 2-6 introduces a number of other features of DL:

Results. The table produced in the process of matching is returned as a result of the procedure. A procedure can have any number of results, which can be any kinds of objects. Part of the definition is a statement of what kind of things the results will be. If the procedure fails, no results are returned. A step calling for a 'Return' causes an immediate stopping of the procedure, just like a 'Succeed' or 'Fail.'

Working structures. During the process of matching, the procedure makes use of a table (one of the primitive objects of DL) to keep track of what word has matched what variable. This is an example of a *working structure.* In this case, it is returned as a result, but a procedure can define and make use of any number of additional structures that are not returned.

Complex conditionals. The nesting of 'If...then...' statements illustrates how one logical expression can be used as part of another. The alignment of the lines in an outline form, along with the little boxes, is DL's way of indicating just how they are structured. Appendix A gives more details both on logical expressions and on the use of outline form.

Generate a sentence from a pattern and bindings

Purpose: Produce a sentence using the result of a match to fill in the blanks of an output pattern

Inputs: a *variable pattern* and a table whose keys are integers and entries are words 2-6

Working Structures:

Response: a sequence of words, initially empty

Results: the completed response

Basic Method:

- Step through elements of the pattern, doing:
  - If the pattern element is:
    - a word, then add it to the end of the response.
    - an integer, then:
      - If there is an entry for it in the table, add the entry to the end of the response. Otherwise fail.
- Return the response.

Figure 2–8. Generate a sentence.

Patterns and generation

In addition to the procedure for recognizing sentences, we can also use patterns and bindings to generate sentences. An early computer program called ELIZA carried on a dialog in which the program simulated a nondirective psychiatrist talking to a patient. To produce the dialog, the program repeatedly read an input sequence, matched it, and then generated and printed a reply using pairs of patterns such as:

My 1 is 2 → How long has your 1 been 2 ?
1 thinks I am 2 → Do you think you are 2 ?

If the person typed *My luck is down,* the program would respond *How long has your luck been down?* Given *Everyone thinks I am crazy,* it would reply *Do you think you are crazy?* The program produced quite convincing dialogs using this simple mechanism, as illustrated by the following dialog (from Weizenbaum, 1966). The computer's responses are in italics:

Men are all alike.
In what way?
They're always bugging me 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.

Of course, ELIZA also produced blunders, since it had no analysis of meaning beyond the recognition of simple patterns, and for many inputs it had no matching patterns and responded with phrases like *Tell me more* and *Please go on*. We could obtain a simplified version of the program by changing our overall search procedure (Figure 2–2) to use a set of pairs, each consisting of a recognition pattern and a generation pattern. Responses would be generated using the procedure of Figure 2–8 on the preceding page. In the actual ELIZA program, variables could be matched against sequences of words, rather than just against single words. This further complexity is given as an exercise.

2.2 Word classes

The first thing to do in building a better pattern recognizer is to make it more selective. The pattern *I* _ *to* _ matches many real sentences such as *I want to go,* but it also matches *I elephant to the,* which is not a sentence. A person's knowledge of a language includes a more precise notion of what words can fill in the blanks.

Much of what is taught in elementary school grammar is the identification of *lexical categories,* often called *word classes* or *parts of speech.* Students learn to assign words to categories like *noun, verb,* and *adjective,* based on their intuitions about language structure. With these classes the sentences *Fat giraffes munch leaves* and *Brainy rabbits nibble carrots* can both be described by the single pattern 'ADJECTIVE NOUN VERB NOUN.'

LEXICAL PATTERN

Describes: a pattern whose elements specify lexical categories, as well as specific words to match

A Kind of: *Simple pattern* 2–4

Background: a *dictionary* 2–10

Basic Structure: a sequence, each member of which is either a word, a *lexical category,* or the character '_' 2–10

Predicates:

An element of a pattern matches a word If the element is:
- the character '_', or
- the word, or
- a *lexical category* to which the word belongs. 2–10

Figure 2 9. Lexical pattern.

Figure 2–9 defines a lexical pattern as one whose elements can specify lexical categories, and gives a definition of matching that assumes the language user has a simple *dictionary* (defined in Figure 2–10) listing the classes to which each word belongs. This is indicated as part of the 'background' rather than as an input to the matching procedure, since structures like dictionaries and grammars tend to serve as a fairly permanent common body of knowledge used by many procedures. This is not a firm distinction—the choice of whether to consider something as an input or a background depends on how we are thinking of the structure of the overall system of definitions.

The dictionary

By putting the definition of lexical category inside a definition of dictionary, we indicate that it makes sense for a word to be in a category only with respect to some dictionary—different dictionaries may have different sets of categories that do not correspond to each other in a simple way. A number of problems are ignored in this simplified notion of a dictionary. For example, we do not deal with the relationships between words like *gopher* and *gophers* or *go* and *going.* However, for many computer applications a dictionary not much more complex than this one is sufficient.

One extension to this simple dictionary would be to use word endings to identify the class to which a word belongs. For example, a word ending with *-ly* is likely to be an adverb, while one ending with *-ing* is probably going to be

DICTIONARY

Describes: a table associating word classes with individual words

Basic Structure: a table: each key is a word and each entry is a set of *lexical categories* §

We need to provide for the fact that many words are in more than one category.

Classes:

LEXICAL CATEGORY

Describes: a word class

Predicates:

A word belongs to a lexical category if there is a pair in the dictionary with the word as the key and the category a member of the entry.

Background: a dictionary

Instances: Noun, Verb, Adjective, Preposition,...

Figure 2–10. Dictionary.

a verb. If there is an analysis program that can use knowledge about standard suffixes, then the dictionary does not need to contain *thinking* in addition to the verb *think*.

Some programs reduce the dictionary even further, including in it only the stems of a few hundred very common words. They derive the rest of the categories by guessing, with a high degree of accuracy. Since these programs do not expect to find a stem for every word, they make mistakes with words like *anomaly* and *thing*, whose endings belie their real class. The practicality of such an approach depends on whether occasional mislabeling of a word prevents the rest of the system from handling a sentence. If a system can make flexible use of multiple knowledge sources, it is even possible to guess the class of an unknown word on the basis of analyzing the rest of the sentence. Human language users gain much of their vocabulary this way, as illustrated by our ability to understand (in some sense) sentences like *Twas brillig and the slithy tove...*

In most of our discussion of syntax, we will assume that there is a procedure that operates on the initial input (a typed sequence of characters), using some kind of simple dictionary and pre-processor to determine a sequence of words and assign each word to a set of word classes. However, before going on we will describe some of the problems presented by the facts of natural language.

What is a word?

The first impression of a literate language user is that the notion of *word* is basic and simple. A word is whatever appears between blanks and punctuation marks on a page. Even in spoken language, it seems quite reasonable to attempt to say something 'one word at a time.' As with many features of language, though, a simple definition works for a great majority of the items in the language but also leaves many unsolved problems.

Structural linguists emphasized the need to provide clear formal definitions of the components of linguistic utterances, and in doing this found the need to talk about a unit called a *morpheme*, which corresponded in some but not all ways to our common sense notion of a word. The morpheme is defined to be the basic unit of meaning, and several morphemes can be combined in a single word. New problems come up in trying to formalize the idea of 'unit of meaning,' but to a large extent we can get agreement on the presence of morphemes. The word *computerization* is made up of *compute, -er, -ize,* and *-ation*. In German, the phrase *life insurance company employee* is expressed as the single multi-morpheme word *Lebensversicherungsgesellschaftsangestellter*. Some languages (called *agglutinating languages*) build up complex words to convey some of the same information that we express with syntactic structures in English. In Turkish, for example, much of the verb and preposition structure is 'glued together.' *Kullanilamiyacak* means *it will not be able to be used,* and *cocuklarinizdan* means *from your children*.

In view of phenomena like these, it is clear that our notion of what constitutes a single word cannot depend on our intuitions about how much separate meaning each word should convey. It is also not reliable to depend on the way things are spelled or pronounced. Even in English, which compared to many languages has a clear separation of words, there are examples that cause problems for analysis. The phrases *can not, cannot,* and *can't* share a common meaning. *Can not* is clearly two words, but the others are less certain. The term *contraction* is used for a class of problematic structures such as *we've, wouldn't,* and *she'll,* which behave very much like single words although they are derived from pairs of words.

The native speaker's intuition of what constitutes a word seems to be most closely captured by some scale that combines *external mobility* with *internal stability*. If a combination of morphemes can be shifted together to different places in a sentence, it is more wordlike than one with a fixed position. If it is possible to insert an item between two morphemes, they are not parts of a single word. *Haven't* is more wordlike than *have not,* since we can say things like *have usually not,* but not *have usually n't*.

Within the paradigm of structural linguistics, many scholarly papers dealt with these problems and pointed out the failings of all the simple criteria that had been proposed for separating words. This was one of the dissatisfactions that led to moving away from the structural paradigm. In this book, we give rather little attention to the problems of *morphology,* as the study of morphemes is called. This does not imply that these problems are irrelevant to theories of how people understand and produce language. It is rather an indication that little exploration has been done of how the computational paradigm can add new insights to the large body of knowledge accumulated by the structuralists.

The classification of words

Anyone with a traditional grammar education from high school comes to linguistics with a seriously limited view of word classes. The impression conveyed in most English courses is that there is a well-defined set of *parts of speech* and that every English word can be straightforwardly assigned to one or more of them. This is true in an artificial language (for example, a computer language), where we can invent arbitrary categories, but in natural languages the grouping of words into classes is highly dependent on the purposes of classification.

Common sense ways of classifying words operate along several dimensions that are partially independent. One set of classes is based on meaning, as in the traditional definition, 'a noun names a person, place, thing, or concept.' A second set of criteria is based on the kinds of endings words will take. We can expect, for example, that any word that will accept *-ing* as a suffix is a verb. Another method of classification is based on the set of patterns in which a word can appear. A word that will fit into *It was a very __ year* is an adjective.

Category	Frame
Noun	*I saw a* ___ .
Verb	*The sun will* ___ .
Article	___ *sun will explode.*
Auxiliary	*The sun* ___ *explode.*

Figure 2–11. Distributional frames.

The problem is that these dimensions do not necessarily go together. In any classification system, it is critical to separate distinctions being made on the basis of *function* from those being made on the basis of *form*. For example, how should we classify the word *laughing*? *Laugh* is a verb, which fits into frames such as *They will* ___. *Laughing* is its *progressive* form which fits into verb frames such as *They had been* ___. However, *laughing* also fits into the frame *I saw a* ___ *cow,* which is associated with adjectives like *purple,* and the frame *The* ___ *lasted for days and days,* which fits nouns like *merriment.*

Looking at the form of *laughing* and the way it is derived, we should classify it as a verb. Looking at the way it functions in sentences, we would sometimes call it a verb, sometimes a noun, and sometimes an adjective. A word like *flash* is both a verb and a noun and represents the same event in both cases. Many classes (particularly in languages like English, with weak inflectional systems) do not take suffixes and prefixes that can be systematically used to classify them. Informally, we can say that the same word is used in different classes, depending on its specific form and context. We run into problems when we try to characterize this formally in a way that could be used to provide detailed rules for syntax.

Categories based on distribution. Structural linguists felt that the best criteria for classifying linguistic entities (including words) were *distributional* tests, since these could be objectively applied to samples of a language. Figure 2–11 illustrates a distributional definition for noun, verb, article, and auxiliary, based on finding distinct *frames* in which they can appear.

There are many problems with the simple idea of distribution. In a real language sample, we are very unlikely to see the word string *I saw a recognition* or *Colorless green ideas sleep furiously.* Does this mean that the words are of the wrong classes, or simply that they do not have appropriate meanings? In fact, as we try to make more precise distinctions, we find that very few words have really identical distribution patterns (see the exercises). It is possible to come up with distributional criteria if we are dealing with a computer program that accepts a carefully limited subset of English, but if we attempt to apply the technique to the language as a whole, we bog down in the details.

Categories based on meaning. It is possible instead to adopt a categorization based on meaning. This approach has been proposed primarily for use in computer systems that do not try to analyze the structure of sentences fully. In Chapter 7 we will describe some of these systems, including a program for machine translation designed by Wilks, in which the word classes include categories such as: SUBSTANTIVE-ELEMENT, ENTITY, POTENT-ELEMENT (those that can designate actors), and MARK-ELEMENT (those that can designate items that themselves designate, like thoughts and writings). Some practically oriented computer systems carry this to an extreme, letting the classes of objects they discuss (such as AIRLINE-NAME, AIRPORT-NAME, and PERSON-NAME) serve directly as the word classes. Such systems are said to use a *semantic grammar.*

In many classification systems, it is convenient to introduce a notion of *subcategorization* in order to account for the fact that all members of one class can automatically be members of another. Every human is also animate. Every animate object is a physical object, and so on. This kind of categorization has been applied both in computer models and in more traditional approaches within the structural and generative paradigms. In Section 6.3 we will discuss the issues of subcategorization further as it applies to both words and structures.

The problem with a meaning-based approach is that semantic word classes are not sufficient for analyzing syntactic structures, since meaning categories correspond so poorly to the determination of which words appear in which structures. A person can *go to town* or *return to town,* and it is clear that the verbs *go* and *return* are in the same overall meaning class. It is normal English to say *We were waiting for your return,* but not *We were waiting for your go.* The fact that *return* but not *go* is also a noun cannot be determined from meaning criteria.

Whatever criteria we use for classification, we will find that some word classes are *closed*—they contain a fixed set of members, and additions are extremely rare—while other classes are *open,* and there are mechanisms for generating new members as new concepts need to be expressed. In English, the class containing the words *and, or,* and *nor* is a closed class with fewer than a dozen members while nouns, verbs, adjectives, and adverbs are open classes. We can invent new names for objects, actions, and properties either by producing totally new words or by adding endings to words from other classes. The noun *computer* came from the verb *compute,* and in turn is the basis for the verb *computerize,* which can be further adapted to form an adjective *computerizable.* This process is discussed further in Section B.4.

Traditional word class definitions

As pointed out earlier, there is an established set of classes for English, used in the traditional teaching of grammar, which is not adequate for developing

comprehensive formal accounts of syntax. It nevertheless serves as a basic vocabulary that most linguists use as a starting point, and it is important to be familiar with these classes.

Most of the categories had their origins in studies of Latin grammar in the Middle Ages. In fact, for many years preceding the segment of linguistic history presented in Chapter 1, the study of language in Europe was dominated by analogies with Latin. It is now accepted as obvious by linguists that the categories applying to one language cannot be applied directly to a different language, even one that is related or is a derivative. However, there appear to be some fundamental similarities among the structures of all languages, for instance the presence of categories corresponding roughly to our notions of noun and verb.

There are a number of fascinating questions about the form of a *universal grammar* that would capture these similarities and shed light on the underlying mental structures on which words and categories are based. It must take into account the possibility emphasized by Whorf (in *Language, Thought, and Reality*) that although our language is shaped by our thinking, our thinking is also shaped by our language. He pointed out that the basic conceptual structure of a society may be based on the mechanisms offered by its language for describing objects and events. Unfortunately, such issues are far beyond the scope of what has been explored within the computational paradigm for language, and they will not be pursued in this book.

The following description of English categories makes use of phrases that might appear in a high school grammar book. These phrases are enclosed in quotes to indicate that they are to be taken as rough guides, not formal definitions. This brief summary is intended to provide terminology for the discussions of syntax in this chapter and in Chapter 3. These chapters do not deal with the complexities of full English but present some basic mechanisms using simplified grammars. Section B.4 of Appendix B presents a more systematic set of categories based on the formalisms described in the rest of the book.

Noun. A noun represents a 'person, place, thing, or concept.' Nouns can take endings to represent plural (*bug* → *bugs*) and possessive (*bug* → *bug's*). There is a subclass of nouns called *proper nouns*, which are names for people, places, etc. There is another subclass called *pronouns*, which 'substitute for a noun.' These include *personal pronouns* (*she, he, they, it, we, I, you,...*), *demonstrative* pronouns (*this, that, these, those*), *possessive* pronouns (*my, your, her, his, hers,...*), and *relative* pronouns (*who, which, that,...*), which connect subordinate clauses, as in *The man who sold Gustav the goldfish.*

Verb. A verb 'signals the performance of an action, the occurrence of an event, or the presence of a condition.' Verbs take the largest variety of endings of any English class, including *present participle* (*-ing*), *past tense* (*-ed*) and *past participle* (*-en*). Many verbs have irregular forms instead of the standard endings, for example, *ring, rang, rung; break, broke, broken.* There is a subclass

of *auxiliary verbs* (*will, can, has, is,...*) which precede a main verb in a phrase like *He must have been planning it all along.* A subclass of the auxiliaries is the *modals* (*can, may, must, should,...*). There are a number of *verbal* forms that function as nouns and adjectives. These include the *past participle* (*broken, finished,...*) and the *present participle* (*running, laughing*). When a verb in present participle form is used as a noun (as in '*Laughing makes you live longer*), it is called a *gerund.*

Adjective. Adjectives 'modify nouns or noun equivalents,' as in *a contented child* or *the slow running.* They can also appear as a *complement* following certain verbs, as in *She seems sad.*

Article (Determiner). There is a small closed class of words (*the, a, some, an, this,...*) which precede nouns in standard noun group structures. We will divide this class into a number of important subclasses later (as described in Appendix B).

Preposition. Prepositions 'relate a noun or noun equivalent to the rest of the sentence.' They are a fairly small closed class (*in, on, until, by, at,...*).

Adverb. Adverbs 'modify other parts of a sentence or the sentence as a whole.' This is the leftover category—anything that doesn't have another class gets called an adverb. It includes words that modify adjectives (*very, somewhat,...*), words that specify time (*usually, now,...*), place (*here, there, somewhere,...*), manner (*quickly, easily,...*), reasoning (*thus, so,...*), and a variety of other stray but useful things (*only, not, even,...*). They are almost always treated as a set of special cases, except for the adverbs generated from adjectives by adding *-ly*, which specify manner.

Conjunctive adverb. One of the adverb classes is used to connect major parts of sentences. Although some such adverbs (*therefore, so,...*) may also introduce a clause that stands alone, there are others (*while, because, unless,...*) that are limited to introducing an embedded clause. These are often called *subordinate conjunctions.*

Coordinate conjunction. Conjunctions, as their name implies, join things. They are one of the hardest things in the grammar to handle well, and their treatment is a major issue of syntax. They form a small closed class (*and, or, but,...*). The conjunctive adverbs can be thought of as a kind of conjunction as well as a kind of adverb.

Interjection. When all is said and done, there are a few stray things like *oh, hey, ouch,* (and many others unprintable) that are interjected into utterances as a way of conveying reactions. This class is rarely handled in computer systems, since people tend not to use them in written language, and current computer programs are not able to make use of the kinds of meanings they convey.

2.3 Transition networks

This section generalizes the idea of a pattern in a way that brings it much closer to the intuitive sense that sentences are 'cut from the same pattern.' The concept of phrase structure in Chapter 3 carries this extension a step further.

In describing the structure of sentences such as *Fat giraffes cavort* and *Wooly bears love sticky honey,* we could use two separate patterns: 'ADJECTIVE NOUN VERB' and 'ADJECTIVE NOUN VERB ADJECTIVE NOUN.' However, this misses the fact that they have an initial section in common. It seems more economical to have some kind of notation (such as brackets) indicating an optional element and to combine them into a single pattern 'ADJECTIVE NOUN VERB {ADJECTIVE NOUN}.' Similarly, we notice that some elements can be repeated an indefinite number of times. If we want to represent the structures of *bears, wooly bears, ferocious wooly bears, hungry ferocious wooly bears,* etc., it is unsatisfactory to have a set of separate patterns:

Noun
Adjective Noun
Adjective Adjective Noun
Adjective Adjective Adjective Noun

Instead, we want another notation to represent optional repetition (we will use a '*') so we can combine them into a single pattern 'ADJECTIVE* NOUN.' By convention, the symbol '*' means 'zero or more repetitions,' so it covers the first pattern as well. In addition, we note that in place of a noun with preceding adjectives we might have a pronoun, as in *He loves honey.* We can use the logical symbol for 'or' ('∨') to represent alternatives and parentheses to indicate grouping, so that for example '(A ∨ B)*' means a repeated sequence, each member of which is either A or B, while 'A ∨ B*' is a choice between a single A and an indefinitely long sequence of B's. The choice between a noun with adjectives and a pronoun is expressed as 'PRONOUN ∨ (ADJECTIVE* NOUN).' Finally, combining all of these we can express a wide variety of structures in a single pattern:

(Pronoun ∨ (Adjective* Noun)) Verb {(Pronoun ∨ (Adjective* Noun))}

This kind of algebraic notation has been developed as part of the theory of formal languages. The specific form introduced here is the *regular language* formalism developed by Kleene. It is quite useful for proving things about languages and about the kinds of machines that can recognize them. However, from the point of view of specifying procedures for doing recognition and parsing, this kind of pattern is complex. It is difficult to deal with a pattern that contains in it symbols like '(' and '∨' that do not match elements in the word string themselves but affect the matching of the other symbols.

Fortunately, there is a more usable notation that gives us the same power, called *state transition diagrams* or *finite state machines* in *automata theory,* a branch of mathematics that serves as a basis for much of the theory of computation. We will use *transition networks* (as these same structures are often called in computational linguistics) as a way of organizing processes both for the production and the analysis of linguistic structures. The rest of this chapter describes these networks and explains how they are used. The correspondence between networks and regular language expressions is covered in textbooks on automata theory and is explored in some of the exercises.

A transition network consists of a set of *states,* connected by *arcs.* Each arc represents a *transition* between two states. In figures, states will be represented by circles and arcs by lines between them, with an arrowhead indicating the direction of the arc. A transition network can be viewed as a pattern for recognizing or generating sequences of words. In both generating and recognizing, the process follows the form of the net in a step-by-step way—each transition along an arc corresponds to a single word in the sequence. The pattern is used by 'stepping through' the transitions from state to state, following the arrows. The stepping must begin in an *initial state* (one marked by a small arrow) and end in a *terminal state* (indicated by a double circle). Figure 2–12 is an example of a transition network, matching the same set of sentence patterns described in the long regular expression above. The letters in the circles for the states do not affect the way the network is used but are there for use in describing it.

The pattern corresponding to *fat wooly bears love honey* starts in state a and goes over a series of arcs 'a –ADJECTIVE→ a –ADJECTIVE→ a –NOUN→ b –VERB→ c –NOUN→ d,' while *they gobble it* would be 'e –PRONOUN→ b –VERB→ f –PRONOUN→ d.' Note that two verb arcs are needed, since if there were simply a pronoun arc connecting states c and d, the network would allow sequences of adjectives followed by a pronoun.

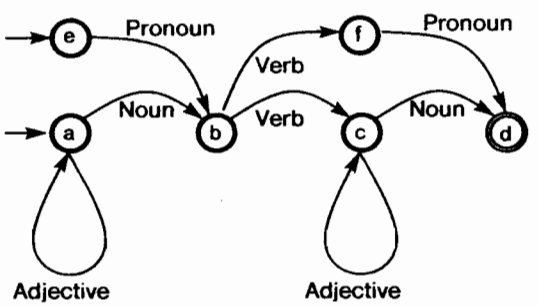

Figure 2-12. A simple transition network.