Language as a Cognitive Process

Volume I: Syntax

Terry Winograd
Stanford University

© 1983

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

Exercises for Section 1.4

1.9 As we will see in later chapters, it is much more difficult for computers to process ordinary colloquial language than to deal with carefully written text. For what applications is it important to go beyond the limitations of fully grammatical, well organized text?

1.10 What are some potentially undesirable effects of developing and using computer systems for natural language?

1.11 What kinds of problems would a computer 'text critiquing' system be able to detect most easily? What kinds would present more difficulty?

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

_ is _.
_ has _.
_ owns _.
Where is _?
Is _ part of _?
Does _ own _?
_ is _ part of _.
There are _ on _.
_ is to the left of _.
Is _ to the left of _?
_ is to the right of _.
Is _ to the right of _?

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

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.

P A T T E R N

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

Describes: a pattern with every element specified

A Kind of: *Simple pattern* 2–4

Basic Structure: a sequence of words

Predicates:

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

OPEN PATTERN

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

A Kind of: *Simple pattern* 2–4

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

Describes: a pattern whose elements are words and variables

A Kind of: *Pattern* 2–3

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   2 6
  are words
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.

Stack	Word	Current Arc	Actions
$[1\text{: } {}_a\text{Det}_b\ {}_b\text{Adj}_b\ {}_b\text{Noun}_c]$			
	the	${}_a\text{Det}_b$	Match
$[2\text{: } {}_b\text{Adj}_b\ {}_b\text{Noun}_c][1\text{: } {}_b\text{Adj}_b\ {}_b\text{Noun}_c]$			
	little	${}_b\text{Adj}_b$	Match
$[3\text{: } {}_b\text{Adj}_b\ {}_b\text{Noun}_c][2\text{: } {}_b\text{Noun}_c][1\text{: } {}_b\text{Adj}_b\ {}_b\text{Noun}_c]$			
	orange	${}_b\text{Adj}_b$	Match
$[4\text{: } {}_b\text{Adj}_b\ {}_b\text{Noun}_c][3\text{: } {}_b\text{Noun}_c][2\text{: } {}_b\text{Noun}_c][1\text{: } {}_b\text{Adj}_b\ {}_b\text{Noun}_c]$			
	ducks	${}_b\text{Adj}_b$	No match
$[4\text{: } {}_b\text{Noun}_c][3\text{: } {}_b\text{Noun}_c][2\text{: } {}_b\text{Noun}_c][1\text{: } {}_b\text{Adj}_b\ {}_b\text{Noun}_c]$			
	ducks	${}_b\text{Noun}_c$	Match and pop
$[5\text{: } {}_c\text{Verb}_d\ {}_c\text{Verb}_e][3\text{: } {}_b\text{Noun}_c][2\text{: } {}_b\text{Noun}_c][1\text{: } {}_b\text{Adj}_b\ {}_b\text{Noun}_c]$			
	swallow	${}_c\text{Verb}_d$	Match
$[6\text{: } {}_d\text{Det}_e][5\text{: } {}_c\text{Verb}_e][3\text{: } {}_b\text{Noun}_c][2\text{: } {}_b\text{Noun}_c][1\text{: } {}_b\text{Adj}_b\ {}_b\text{Noun}_c]$			
	flies	${}_d\text{Det}_e$	No match and pop
$[5\text{: } {}_c\text{Verb}_e][3\text{: } {}_b\text{Noun}_c][2\text{: } {}_b\text{Noun}_c][1\text{: } {}_b\text{Adj}_b\ {}_b\text{Noun}_c]$			
	swallow	${}_c\text{Verb}_e$	Match and pop
$[6\text{: } {}_e\text{Adj}_e\ {}_e\text{Noun}_f][3\text{: } {}_b\text{Noun}_c][2\text{: } {}_b\text{Noun}_c][1\text{: } {}_b\text{Adj}_b\ {}_b\text{Noun}_c]$			
	flies	${}_e\text{Adj}_e$	No match
$[6\text{: } {}_e\text{Noun}_f][3\text{: } {}_b\text{Noun}_c][2\text{: } {}_b\text{Noun}_c][1\text{: } {}_b\text{Adj}_b\ {}_b\text{Noun}_c]$			
	flies	${}_e\text{Noun}_f$	Match and succeed

Figure 2-22. A backtracking recognition.

In fact, (as explored in the exercises) even relatively simple networks can produce large amounts of backtracking. In particular, it is important to note that when backtracking happens, there are no records left around about any of the work that was done beyond that position, so it must be redone as the process goes forward again.

Further Reading for Chapter 2

Pattern-based computer programs. ELIZA is described in Weizenbaum (1966). A program that later developed the pattern concept in a much more elaborate way was Colby's (1976) PARRY, which played the role of a paranoid mental patient instead of a psychiatrist! Raphael's SIR program is described along with several other early natural language programs in *Semantic Information Processing,* edited by Minsky (1967).

Word classes. For more information on word classes from a traditional linguistic point of view, see any of the books listed in Chapter 1 as references for the history of linguistics. In particular, Chapter 5 of Lyons's *Introduction to Theoretical Linguistics* deals with many of these issues. For an encyclopedic account of word classes, see Quirk et al., *A Grammar of Contemporary English.*

Networks and regular languages. There are a number of texts describing finite state machines (transition networks) and formal languages from a mathematical point of view. Minsky's *Computation: Finite and Infinite Machines* is easy reading, while Hopcroft and Ullman's *Formal Languages and Their Relation to Automata* is more comprehensive and serves as a standard textbook for many courses. For a discussion of algorithms in general that will serve as background for the formalization of computational processes, see Knuth, *The Art of Computer Programming,* Volume 1.

Backtracking. Backtracking was proposed as a programming technique by Golomb and Baumert (1965). It is a standard feature of programming languages designed for use in artificial intelligence, as described in Bobrow and Raphael's 1974 survey of 'New programming languages for artificial intelligence research.' For a discussion of the problems inherent in chronological backtracking and a description of some more sophisticated alternatives, see Nilsson (1980).

Exercises for Chapter 2

Exercises for Section 2.1

2.1 It was pointed out in the text that the procedures described in Section 2.1 could be modified to use recognition-response pairs like those of ELIZA. Write a new version of the procedure in Figure 2-2 which takes a sequence of words and a set of 'pairs,' each containing two patterns with variables: a *recognition* pattern and a *response* pattern. It should find a pair whose recognition pattern matches the sequence and generate a response from the response pattern using the same bindings.

2.2 In Exercise 2.13, we will create a procedure for matching a pattern with variables that allows a single variable to match a sequence of words. For example, applying the recognition-response pair: 'My 1 is 2 → What if your 1 were not 2' to the sentence *My left arm is about to fall off* would produce *What if your left arm were not about to fall off*. Assuming such an extension, analyze carefully what the responses would be to the following inputs:

My head is on my shoulders
My problem is that you hate me
My problem is how to pay you
My brother said your car is bigger than mine
My job is working in the mine

Describe the changes to the procedure that would be needed to produce responses that are syntactically appropriate (don't worry about their therapeutic appropriateness!). What kinds of problems stand in the way of a general solution within the framework of pattern-matching? The algorithm actually used by ELIZA gets three of these examples right and two of them wrong.

Exercises for Section 2.2

2.3 Give some arguments on both sides of the issue as to whether the English possessive *'s* should be considered a separate word or not. Consider sentences like *I saw the man you met in Ankara last year's brother.*

2.4 Consider the three linguistic frames:

1) *Miss Muffett __ to eat whey with curd.*
2) *Sybill __ her man to be a good cook.*
3) *Stu was __ to be on time.*

a) Classify the following verbs according to which of these frames they can fill: *asked, preferred, condescended, believed, promised, wanted, tried, considered, accepted, forced, expected.*

b) Find frames that show that no two of the above verbs have the identical distribution.

c) Can you find a verb which, as far as you can tell, has the same distribution as 1) *promise,* 2) *ask,* 3) *believe?*

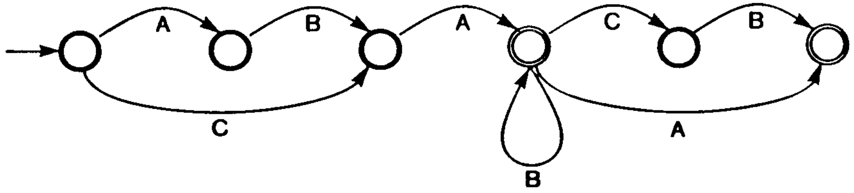
2.5 Try to identify every word in the following paragraph by its traditional word class.

> *It is of course true that quality is much more difficult to 'handle' than quantity, just as the exercise of judgment is a higher function than the ability to count and calculate. Quantitative differences can be more easily grasped and certainly more easily defined than qualitative differences; their concreteness is beguiling and gives them the appearance of scientific precision, even when this precision has been purchased by the suppression of vital differences of quality. The great majority of economists is still pursuing the absurd ideal of making their 'science' as scientific and precise as physics, as if there were no qualitative differences between mindless atoms and men made in the image of God.*

2.6 Explain how the concept of closed and open word classes relates to the fact that it is relatively easy to deal with some of the sex biases of language by creating new words such as *repairperson* and *congressperson,* but that it is extremely difficult to find an appropriate word to use in the sentence *When the chairperson brings the meeting to order, __ must pound the gavel with vigor.*

Exercises for Section 2.3

2.7 Which of the following sequences can be described (recognized or generated) by this transition network:



(1) ab (2) ca (3) cb (4) cabbb (5) ababa (6) bacb (7) cabcb (8) ababcbc

2.8 Write a regular expression corresponding to the transition network in the previous exercise. Use the standard convention for regular languages that an asterisk means *zero or more* repetitions and the superscript '+' is used to mean *one or more.* For example, 'a*b' matches the expressions 'b, ab, aab, aaab,...' while 'a^+b' matches all but the first of them.

2.9 Draw a transition network corresponding to the regular expression:

$$(a \vee (b\ a\ b))^* (b^* \vee a)$$

2.10 Draw a transition network that will match the expressions for time of day in English, such as *one thirty-two and twelve seconds, half past three,* and *six fifteen a.m.*. The basic lexical categories will include things like:

Hour-Number: *one, two,... , twelve*
Minute-Number: *one, two,... , fifty-nine*
Fraction: *quarter, half*

There will also be arcs whose labels are specific words such as *to, past, after, till, o'clock, a.m.,* and *p.m.*

2.11 (*For programmers*) If the definitions of this chapter were implemented in a straightforward way, there would be a data structure for a transition network that contained structures for arcs and states. Networks can be compiled into a more efficient form in which the states of the network correspond to states of the program. Describe or demonstrate how a transition network can be compiled so that a state corresponds to a location (e.g., a GO TO label) in a program.

Exercises for Section 2.4

2.12 If we had chosen a different convention for ordering arcs, the backtracking process described in Figure 2–22 would have gone differently. Assume reverse alphabetical order on each arc's starting vertex, label, and ending vertex. That is, the arcs of the network in Figure 2 16 are ordered: ${}_{e}\mathrm{NOUN}_{f}$, ${}_{e}\mathrm{ADJ}_{e}$, ${}_{d}\mathrm{DET}_{e}$, ${}_{c}\mathrm{VERB}_{e}$, ${}_{c}\mathrm{VERB}_{d}$, ${}_{b}\mathrm{NOUN}_{c}$, ${}_{b}\mathrm{ADJ}_{b}$, ${}_{a}\mathrm{DET}_{b}$. Generate a trace like that of Figure 2–22 using the same input sentence.

2.13 If we want variables in a pattern to match sequences of words, there is a problem in knowing what to do as we proceed from left to right. For example, the pattern '1 X Y' will match the sequence 'YXY' with the variable matching the first 'Y' and will match 'YXYXY' if the variable matches 'YXY'. If the procedure simply takes the element of the pattern and sequence in order, it will not be able to decide whether the first 'X' should be included in the variable or matched against the 'X' in the pattern.

a) Write a nondeterministic schema for matching a pattern with variables to a sequence of words, which allows a variable to match any sequence of *one or more* words in the input.

b) Write a backtracking procedure that deterministically matches the same kinds of patterns.

2.14 In introducing the problem of dealing with choices in networks, we commented that a network would be deterministic (involve no choices) if it had a single initial state, if no two arcs with the same starting state had the same label, and if no two arcs with different labels could match the same word. In fact, any transition network of the kind defined in Figure 2-13 can be used to produce a deterministic network that is *weakly equivalent* in that it will accept and reject exactly the same inputs. The new network will have a state for each subset of the states in the original and each of its arcs will match a particular word, not a lexical category. Describe how to derive this network from the original network and dictionary.

2.15 (*For programmers*) In a recursive language such as LISP, it is possible to use the internal stack on which variables are bound as the means of keeping the stack for backtracking. Write a recursive procedure for recognition with a transition network in which the stack is maintained this way.