FUTURE VISION BIE

One Stop for All Study Materials

& Lab Programs

Future Vision

By K B Hemanth Raj

Scan the QR Code to Visit the Web Page

Or

Visit : https://hemanthrajhemu.github.io

Gain Access to All Study Materials according to VTU, Currently for CSE – Computer Science Engineering...

Join Telegram to get Instant Updates: https://bit.ly/2GKiHnJ

Contact: MAIL: futurevisionbie@gmail.com

INSTAGRAM: www.instagram.com/hemanthraj_hemu/

INSTAGRAM: www.instagram.com/futurevisionbie/

CHAPTER

15

NATURAL LANGUAGE PROCESSING

The man who knows no foreign language knows nothing of his mother tongue.

—**Johann Wolfgang von Geoethe**
(1749-1832), German poet. novelist, playwright and philosopher

Language is meant for communicating about the world. By studying language, we can come to understand more about the world. We can test our theories about the world by how well they support our attempt to understand language. And, if we can succeed at building a computational model of language, we will have a powerful tool for communicating about the world. In this chapter, we look at how we can exploit knowledge about the world, in combination with linguistic facts, to build computational natural language systems.

Throughout this discussion, it is going to be important to keep in mind that the difficulties we will encounter do not exist out of perversity on the part of some diabolical designer. Instead, what we see as difficulties when we try to analyze language are just the flip sides of the very properties that make language so powerful. Figure 15.1 shows some examples of this. As we pursue our discussion of language processing, it is important to keep the good sides in mind since it is because of them that language is significant enough a phenomenon to be worth all the trouble.

By far the largest part of human linguistic communication occurs as speech. Written language is a fairly recent invention and still plays a less central role than speech in most activities. But processing written language (assuming it is written in unambiguous characters) is easier, in some ways, than processing speech. For example, to build a program that understands spoken language, we need all the facilities of a written language understand as well as enough additional knowledge to handle all the noise and ambiguities of the audio signal.[1] Thus it is useful to divide the entire language-processing problem into two tasks:

- Processing written text, using lexical, syntactic, and semantic knowledge of the language as well as the required real world information
- Processing spoken language, using all the information needed above plus additional knowledge about phonology as well as enough added informal ion to handle the further ambiguities that arise in speech

[1]Actually, in understanding spoken language, we take advantage of clues, such as intonation and the presence of pauses, to which we do not have access when we read. We can make the task of a speech-understanding program easier by allowing it, too, to use these clues, but to do so, we must know enough about them to incorporate into the program knowledge of how to use them.

The Problem: English sentences are incomplete descriptions of the information that they are intended to convey:

Some dogs are outside.	I called Lynda to ask her to the movies. She said she'd love to go.
↓	↓
Some dogs are on the lawn.	She was home when I called.
Three dogs are on the lawn.	She answered the phone.
Rover, Tripp, and Spot are on the lawn.	I actually asked her.

The Good Side: Language allows speakers to be as vague or as precise as they like. It also allows speakers to leave out things they believe their hearers already know.

The Problem: The same expression means different things in different contexts:

Where's the water? (in a chemistry lab, it must be pure)
Where's the water? (when you are thirsty, it must be potable)
Where's the water? (dealing with a leaky roof, it can be filthy)

The Good Side: Language lets us communicate about an infinite world using a finite (and thus learnable) number of symbols.

The Problem: No natural language program can be complete because new words, expressions, and meanings can be generated quite freely:

I'll fax it to you.

The Good Side: Language can evolve as the experiences that we want to communicate about evolve.

The Problem: There are lots of ways to say the same thing:

Mary was born on October 11.
Mary's birthday is October 11.

The Good Side: When you know a lot, facts imply each other. Language is intended to be used by agents who know a lot.

Fig. 15.1 *Features of Language That Make It Both Difficult and Useful*

In Chapter 14 we described some of the issues that arise in speech understanding, and in Section 21.2.2 we return to them in more detail. In this chapter, though, we concentrate on written language processing (usually called simply *natural language processing).*

Throughout this discussion of natural language processing, the focus is on English. This happens to be convenient and turns out to be where much of the work in the field has occurred. But the major issues we address are common to all natural languages, In fact, the techniques we discuss are particularly important in the task of translating from one natural language to another.

Natural language processing includes both understanding and generation, as well as other tasks such as multilingual translation. In this chapter we focus on understanding, although in Section 15.5 we will provide some references to work in these other areas.

15.1 INTRODUCTION

Recall that in the last chapter we defined understanding as the process of mapping from an input form into a more immediately useful form. It is this view of understanding that we pursue throughout this chapter. But it is useful to point out here that there is a formal sense in which a language can be defined simply as a set of strings without reference to any world being described or task to be performed. Although some of the ideas that have come out of this formal study of languages can be exploited in parts of the understanding process, they are only the beginning. To get the overall picture, we need to think of language as a pair (source language,

target representation), together with a mapping between elements of each to the other. The target representation will have been chosen to be appropriate for the task at hand. Often, if the task has clearly been agreed on and the details of the target representation are not important in a particular discussion, we talk just about the language itself, but the other half of the pair is really always present.

One of the great philosophical debates throughout the centuries has centered around the question of what a sentence means. We do not claim to have found the definitive answer to that question. But once we realize that understanding a piece of language involves mapping it into some representation appropriate to a particular situation, it becomes easy to see why the questions "What is language understanding?" and "What does a sentence mean?" have proved to be so difficult to answer. We use language in such a wide variety of situations that no single definition of understanding is able to account for them all. As we set about the task of building computer programs that understand natural language, one of the first things we have to do is define precisely what the underlying task is and what the target representation should look like. In the rest of this chapter, we assume that our goal is to be able to reason with the knowledge contained in the linguistic expressions, and we exploit a frame language as our target representation.

15.1.1 Steps in the Process

Before we go into detail on the several components of the natural language understanding process, it is useful to survey all of them and see how they fit together. Roughly, we can break the process down into the following pieces:

- Morphological Analysis—Individual words are analyzed into their components, and nonword tokens, such as punctuation, are separated from the words.
- Syntactic Analysis—Linear sequences of words are transformed into structures that show how the words relate to each other. Some word sequences may be rejected if they violate the language's rules for how words may be combined. For example, an English syntactic analyzer would reject the sentence "Boy the go the to store."
- Semantic Analysis—The structures created by the syntactic analyzer are assigned meanings. In other words, a mapping is made between the syntactic structures and objects in the task domain. Structures for which no such mapping is possible may be rejected. For example, in most universes, the sentence "Colorless green ideas sleep furiously" [Chomsky, 1957] would be rejected as *semantically anomolous.*
- Discourse Integration—The meaning of an individual sentence may depend on the sentences that precede it and may influence the meanings of the sentences that follow it. For example, the word "it" in the sentence, "John wanted it," depends on the prior discourse context, while the word "John" may influence the meaning of later sentences (such as, "He always had.")
- Pragmatic Analysis—The structure representing what was said is reinterpreted to determine what was actually meant. For example, the sentence "Do you know what time it is?" should be interpreted as a request to be told the time.

The boundaries between these five phases are often very fuzzy. The phases are sometimes performed in sequence, and they are sometimes performed all at once. If they are performed in sequence, one may need to appeal for assistance to another. For example, part of the process of performing the syntactic analysis of the sentence "Is the glass jar peanut butter?" is deciding how to form two noun phrases out of the four nouns at the end of the sentence (giving a sentence of the form "Is the *x y*?"). All of the following constituents are syntactically possible: glass, glass jar, glass jar peanut, jar peanut butter, peanut butter, butter. A syntactic processor on its own has no way to choose among these, and so any decision must be made by appealing to some model of the world in which some of these phrases make sense and others do not. If we do this, then we get a syntactic structure in which the constituents "glass jar" and "peanut butter" appear. Thus although it is

often useful to separate these five processing phases to some extent, they can all interact in a variety of ways, making a complete separation impossible.

Specifically, to make the overall language understanding problem tractable, it will help if we distinguish between the following two ways of decomposing a program:

- The processes and the knowledge required to perform the task
- The global control structure that is imposed on those processes

In this chapter, we focus primarily on the first of these issues. It is the one that has received the most attention from people working on this problem. We do not completely ignore the second issue, although considerably less of substance is known about it. For an example of this kind of discussion that talks about interleaving syntactic and semantic processing, see Lytinen [1986].

With that caveat, let's consider an example to see how the individual processes work. In this example, we assume that the processes happen sequentially. Suppose we have an English interface to an operating system and the following sentence is typed:

I want to print Bill's .init file.

Morphological Analysis

Morphological analysis must do the following things:

- Pull apart the word "Bill's" into the proper noun "Bill" and the possessive suffix "'s"
- Recognize the sequence ".init" as a file extension that is functioning as an adjective in the sentence

In addition, this process will usually assign syntactic categories to all the words in the sentence. This is usually done now because interpretations for affixes (prefixes and suffixes) may depend on the syntactic category of the complete word. For example, consider the word "prints." This word is either a plural noun (with the "-s" marking plural) or a third person singular verb (as in "he prints"), in which case the "-s" indicates both singular and third person. If this step is done now, then in our example, there will be ambiguity since "want," "print," and "file" can all function as more than one syntactic category.

Syntactic Analysis

Syntactic analysis must exploit the results of morphological analysis to build a structural description of the sentence. The goal of this process, called *parsing*, is to convert the flat list of words that forms the sentence into a structure that defines the units that are represented by that fiat list. For our example sentence, the result of parsing is shown in Fig. 15.2. The details of this representation are not particularly significant; we describe alternative versions of them in Section 15.2. What is important here is that a flat sentence has been converted into a hierarchical structure and that that structure has been designed to correspond to sentence units (such as noun phrases) that will correspond to meaning units when semantic analysis is performed. One useful thing we have done here, although not all syntactic systems do, is create a set of entities we call *reference markers*. They are shown in parentheses in the parse tree. Each one corresponds to some entity that has been mentioned in the sentence. These reference markers are useful later since they

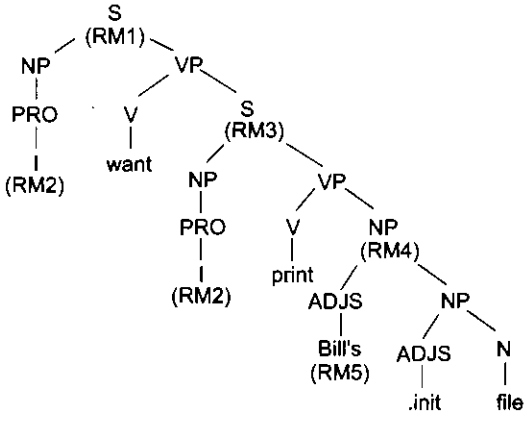


Fig. 15.2 *The Result of Syntactic Analysis of "I want to print Bill's .init file."*

provide a place in which to accumulate information about the entities as we get it. Thus although we have not tried to do semantic analysis (i.e., assign meaning) at this point, we have designed our syntactic analysis process so that it will find constituents to which meaning can be assigned.

Semantic Analysis

Semantic analysis must do two important things:

- It must map individual words into appropriate objects in the knowledge base or database.
- It must create the correct structures to correspond to the way the meanings of the individual words combine with each other.

For this example, suppose that we have a frame-based knowledge base that contains the units shown in Fig. 15.3. Then we can generate a partial meaning, with respect to that knowledge base, as shown in Fig. 15.4. Reference marker *RM1* corresponds to the top-level event of the sentence. It is a wanting event in which the speaker (denoted by "I") wants a printing event to occur in which the same speaker prints a file whose extension is ".init" and whose owner is Bill.

User
  isa: *Person*
  * *login-name*: must be <string>

User068
  instance: *User*
  login-name: *Susan-Black*

User073
  instance: *User*
  login-name: *Bill-Smith*

F1
  instance: *File-Struct*
  name: stuff
  extension: .init
  owner: *User073*
  in-directory: /wsmith/

File-Struct
  isa: *Information-Object*

Printing
  isa: *Physical-Event*
  * *agent*: must be <animate or program>
  * *object*: must be <information-object>

Wanting
  isa: *Mental-Event*
  * *agent*: must be <animate>
  * *object*: must be <state or event>

Commanding
  isa: *Mental-Event*
  * *agent*: must be <animate>
  * *performer*: must be <animate or program>
  * *object*: must be <event>

This-System
  instance: *Program*

Fig. 15.3 *A Knowledge Base Fragment*

RM1		{the whole sentence}
instance :	*Wanting*	
agent :	*RM2*	{I}
object :	*RM3*	{a printing event}
RM2		{I}
RM3		{a printing event}
instance :	*Printing*	
agent :	*RM2*	{I}
object :	*RM4*	{Bill's .init file}
RM4		{Bill's .init file}
instance :	*File-Struct*	
extension :	.init	
owner :	*RM5*	{Bill}
RM5		{Bill}
instance :	*Person*	
first-name :	Bill	

Fig. 15.4 *A Partial Meaning for a Sentence*

Discourse Integration

At this point, we have figured out what kinds of things this sentence is about. But we do not yet know which specific individuals are being referred to. Specifically, we do not know to whom the pronoun "I" or the proper noun "Bill" refers. To pin down these references requires an appeal to a model of the current discourse context, from which we can learn that the current user (who typed the word "I") is *User068* and that the only person named "Bill" about whom we could be talking is *User073*. Once the correct referent for Bill is known, we can also determine exactly which file is being referred to: *F1* is the only file with the extension ".init" that is owned by Bill.

Pragmatic Analysis

We now have a complete description, in the terms provided by our knowledge base, of what was said. The final step toward effective understanding is to decide what to do as a result. One possible thing to do is to record what was said as a fact and be done with it. For some sentences, whose intended effect is clearly declarative, that is precisely the correct thing to do. But for other sentences, including this one, the intended effect is different. We can discover this intended effect by applying a set of rules that characterize cooperative dialogues. In this example, we use the fact that when the user claims to want something that the system is capable of performing, then the system should go ahead and do it. This produces the final meaning shown in Fig. 15.5.

Meaning		
	instance :	*Commanding*
	agent :	*User068*
	performer :	*This-System*
	object :	*P27*
P27		
	instance :	*Printing*
	agent :	*This-System*
	object :	*F1*

Fig. 15.5 *Representing the Intended Meaning*

The final step in pragmatic processing is to translate, when necessary, from the knowledge-based representation to a command to be executed by the system. In this case, this step is necessary, and we see that the final result of the understanding process is

lpr /wsr...th/stuff.txt

where "lpr" is the operating system's file print command.

Summary

At this point, we have seen the results of each of the main processes that combine to form a natural language system. In a complete system, all of these processes are necessary in some form. For example, it may have seemed that we could have skipped the knowledge-based representation of the meaning of the sentence since the final output of the understanding system bore no relationship to it. But it is that intermediate knowledge-based representation to which we usually attach the knowledge that supports the creation of the final answer.

All of the processes we have described are important in a complete natural language understanding system. But not all programs are written with exactly these components. Sometimes two or more of them are collapsed, as we will see in several sections later in this chapter. Doing that usually results in a system that is easier to build for restricted subsets of English but one that is harder to extend to wider coverage. In the rest of this chapter we describe the major processes in more detail and talk about some of the ways in which they can be put together to form a complete system.

15.2 SYNTACTIC PROCESSING

Syntactic processing is the step in which a flat input sentence is converted into a hierarchical structure that corresponds to the units of meaning in the sentence. This process is called *parsing*. Although there are natural language understanding systems that skip this step (for example, see Section 15.3.3), it plays an important role in many natural language understanding systems for two reasons:

- Semantic processing must operate on sentence constituents. If there is no syntactic parsing step, then the semantics system must decide on its own constituents. If parsing is done, on the other hand, it constrains the number of constituents that semantics can consider. Syntactic parsing is computationally less expensive than is semantic processing (which may require substantial inference). Thus it can play a significant role in reducing overall system complexity.
- Although it is often possible to extract the meaning of a sentence without using grammatical facts, it is not always possible to do so. Consider, for example, the sentences
  - The satellite orbited Mars.
  - Mars orbited the satellite.

  In the second sentence, syntactic facts demand an interpretation in which a planet (Mars) revolves around a satellite, despite the apparent improbability of such a scenario.

Although there are many ways to produce a parse, almost all the systems that are actually used have two main components:

- A declarative representation, called a *grammar*, of the syntactic facts about the language
- A procedure, called a *parser*, that compares the grammar against input sentences to produce parsed structures

15.2.1 Grammars and Parsers

The most common way to represent grammars is as a set of production rules. Although details of the forms that are allowed in the rules vary, the basic idea remains the same and is illustrated in Fig. 15.6, which shows a simple context-free, phrase structure grammar for English. Read the first rule as, "A sentence is composed of a noun phrase followed by a verb phrase." In this grammar, the vertical bar should be read as "or." The ε

denotes the empty string. Symbols that are further expanded by rules are called *nonterminal symbols.* Symbols that correspond directly to strings that must be found in an input sentence are called *terminal symbols.*

S → NP VP
NP → the NP1
NP → PRO
NP → PN
NP → NP1
NP1 → ADJS N
ADJS → ε | ADJ ADJS
VP → V
VP → V NP
N → file | printer
PN → Bill
PRO → I
ADJ → short | long | fast
V → printed | created | want

Fig. 15.6 *A Simple Grammar for a Fragment of English*

Grammar formalisms such as this one underlie many linguistic theories, which in turn provide the basis for many natural language understanding systems. Modern linguistic theories include: the government binding theory of Chomsky [1981; 1986], GPSG [Gazdar *et al.,* 1985], LFG [Bresnan, 1982], and categorial grammar [Ades and Steedman, 1982; Oehrle *et al.,* 1987]. The first three of these are also discussed in Sells [1986]. We should point out here that there is general agreement that pure, context-free grammars are not effective for describing natural languages.[2] As a result, natural language processing systems have less in common with computer language processing systems (such as compilers) than you might expect.

Regardless of the theoretical basis of the grammar, the parsing process takes the rules of the grammar and compares them against the input sentence. Each rule that matches adds something to the complete structure that is being built for the sentence. The simplest structure to build is a *parse tree,* which simply records the rules and how they are matched. Figure 15.7 shows the parse tree that would be produced for the sentence "Bill printed the file" using this grammar. Figure 15.2 contained another example of a parse tree, although some additions to this grammar would be required to produce it.

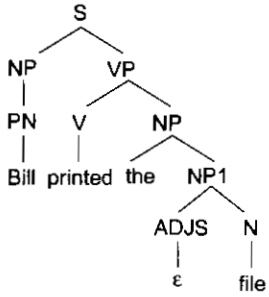


Fig. 15.7 *A Parse Tree for a Sentence*

Notice that every node of the parse tree corresponds either to an input word or to a nonterminal in our grammar. Each level in the parse tree corresponds to the application of one grammar rule. As a result, it should be clear that a grammar specifies two things about a language:

- Its weak generative capacity, by which we mean the set of sentences that are contained within the language. This set (called the set of *grammatical sentences)* is made up of precisely those sentences that can be completely matched by a series of rules in the grammar.
- Its strong generative capacity, by which we mean the structure (or possibly struc- "tures) to be assigned to each grammatical sentence of the language.

[2]There is, however, still some debate on whether context-free grammars are formally adequate for describing natural languages (e.g., Gazdar [1982].)

So far, we have shown the result of parsing to be exactly a trace of the rules that were applied during it. This is not always the case, though. Some grammars contain additional information that describes the structure that should be built. We present an example of such a grammar in Section 15.2.2.

But first we need to look at two important issues that define the space of possible parsers that can exploit the grammars we write.

Top-Down versus Bottom-Up Parsing

To parse a sentence, it is necessary to find a way in which that sentence could have been generated from the start symbol. There are two ways that this can be done:

- *Top-Down Parsing*—Begin with the start symbol and apply the grammar rules forward until the symbols at the terminals of the tree correspond to the components of the sentence being parsed.
- *Bottom-Up Parsing*—Begin with the sentence to be parsed and apply the grammar rules backward until a single tree whose terminals are the words of the sentence and whose top node is the start symbol has been produced.

The choice between these two approaches is similar to the choice between forward and backward reasoning in other problem-solving tasks. The most important consideration is the branching factor. Is it greater going backward or forward? Another important issue is the availability of good heuristics for evaluating progress. Can partial information be used to rule out paths early? Sometimes these two approaches are combined into a single method called *bottom-up parsing with top-down filtering*. In this method, parsing proceeds essentially bottom-up (i.e., the grammar rules are applied backward). But using tables that have been precomputed for a particular grammar, the parser can immediately eliminate constituents that can never be combined into useful higher-level structures.

Finding One Interpretation or Finding Many

As several of the examples above have shown, the process of understanding a sentence is a search process in which a large universe of possible interpretations must be explored to find one that meets all the constraints imposed by a particular sentence.* As for any search process, we must decide whether to explore all possible paths or, instead, to explore only a single most likely one and to produce only the result of that one path as the answer.

Suppose, for example, that a sentence processor looks at the words of an input sentence one at a time, from left to right, and suppose that so far, it has seen:

"Have the students who missed the exam—"

There are two paths that the processor could be following at this point:

- "Have" is the main verb of an imperative sentence, such as
  "Have the students who missed the exam take it today."
- "Have" is an auxiliary verb of an interrogative sentence, such as
  "Have the students who missed the exam taken it today?"

There are four ways of handling sentences such as these:

- *All Paths*—Follow all possible paths and build all the possible intermediate components. Many of the components will later be ignored because the other inputs required to use them will not appear. For example, if the auxiliary verb interpretation of "have" in the previous example is built, it will be discarded if no participle, such as "taken," ever appears. The major disadvantage of this approach is that, because it results in many spurious constituents being built and many deadend paths being followed, it can be very inefficient.

- *Best Path with Backtracking*—Follow only one path at a time, but record, at every choice point, the information that is necessary to make another choice if the chosen path fails to lead to a complete interpretation of the sentence. In this example, if the auxiliary verb interpretation of "have" were chosen first and the end of the sentence appeared with no main verb having been seen, the understander would detect failure and backtrack to try some other path. There are two important drawbacks to this approach. The first is that a good deal of time may be wasted saving state descriptions at each choice point, even though backtracking will occur to only a few of those points. The second is that often the same constituent may be analyzed many times. In our example, if the wrong interpretation is selected for the word "have," it will not he detected until after the phrase "the students who missed the exam" has been recognized. Once the error is detected, a simple backtracking mechanism will undo everything that was done after the incorrect interpretation of "have" was chosen, and the noun phrase will be reinterpreted (identically) after the second interpretation of "have" has been selected. This problem can be avoided using some form of dependency-directed backtracking, but then the implementation of the parser is more complex.
- *Best Path with Patchup*—Follow only one path at a time, but when an error is detected, explicitly shuffle around the components that have already been formed. Again, using the same example, if the auxiliary verb interpretation of "have" were chosen first, then the noun phrase "the students who missed the exam" would be interpreted and recorded as the subject of the sentence. If the word "taker" appears next, this path can simply be continued. But if "take" occurs next, the understander can simply shift components into different slots. "Have" becomes the main verb. The noun phrase that was marked as the subject of the sentence becomes the subject of the embedded sentence "The students who missed the exam take it today." And the subject of the main sentence can be filled in as "you," the default subject for imperative sentences. This approach is usually more efficient than the previous two techniques. Its major disadvantage is that it requires interactions among the rules of the grammar to be made explicit in the rules for moving components from one place to another. The interpreter often becomes *ad hoc,* rather than being simple and driven exclusively from the grammar.
- *Wait and See*—Follow only one path, but rather than making decisions about the function of each component as it is encountered, procrastinate the decision until enough information is available to make the decision correctly. Using this approach, when the word "have" of our example is encountered, it would be recorded as some kind of verb whose function is, as yet, unknown. The following noun phrase would then be interpreted and recorded simply as a noun phrase. Then, when the next word is encountered, a decision can be made about how all the constituents encountered so far should be combined. Although several parsers have used some form of wait-and-see strategy, one, PARSIFAL [Marcus, 1980], relies on it exclusively. It uses a small, fixed-size buffer in which constituents can be stored until their purpose can be decided upon. This approach is very efficient, but it does have the drawback that if the amount of lookahead that is necessary is greater than the size of the buffer, then the interpreter will fail. But the sentences on which it fails are exactly those on which people have trouble, apparently because they choose one interpretation, which proves to be Wrong. A classic example of this phenomenon, called the *garden path sentence,* is

  The horse raced past the barn fell down.

Although the problems of deciding which paths to follow and how to handle backtracking are common to all search processes, they are complicated in the case of language understanding by the existence of genuinely ambiguous sentences, such as our earlier example "They are flying planes." If it is important that not just one interpretation but rather all possible ones be found, then either all possible paths must be followed (which is very expensive since most of them will die out before the end of the sentence) or backtracking must be forced

(which is also expensive because of duplicated computations). Many practical systems are content to find a single plausible interpretation. If that interpretation is later rejected, possibly for semantic or pragmatic reasons, then a new attempt to find a different interpretation can be made.

Parser Summary

As this discussion suggests, there are many different kinds of parsing systems. There are three that have been used fairly extensively in natural language systems:

- Chart parsers [Winograd, 1983], which provide a way of avoiding backup by storing intermediate constituents so that they can be reused along alternative parsing paths.
- Definite clause grammars [Pereira and Warren, 1980], in which grammar rules are written as PROLOG clauses and the PROLOG interpreter is used to perform top-down, depth-first parsing.
- Augmented transition networks (or ATNs) [Woods, 1970]-, in which the parsing process is described as the transition from a start state to a final state in a transition network that corresponds to a grammar of English.

We do not have space here to go into all these methods. In the next section, we illustrate the main ideas involved in parsing by working through an example with an ATN. After this, we look at one way of parsing with a more declarative representation.

15.2.2 Augmented Transition Networks

An augmented transition network (ATN) is a top-down parsing procedure that allows various kinds of knowledge to be incorporated into the parsing system so it can operate efficiently. Since the early use of the ATN in the LUNAR system [Woods, 1973], which provided access to a large database of information on lunar geology, the mechanism has been exploited in many language-understanding systems. The ATN is similar to a finite state machine in which the class of labels that can be attached to the arcs that define transitions between states has been augmented. Arcs may be labeled with an arbitrary combination of the following:

- Specific words, such as "in."
- Word categories, such as "noun."
- Pushes to other networks that recognize significant components of a sentence. For example, a network designed to recognize a prepositional phrase (PP) may include an arc that asks for ("pushes for") a noun phrase (NP).
- Procedures that perform arbitrary tests on both the current input and on sentence components that have already been identified.
- Procedures that build structures that will form part of the final parse.

Figure 15.8 shows an example of an ATN in graphical notation. Figure 15.9 shows the top-level ATN of that example in a notation that a program could read. To see how an ATN works, let us trace the execution of this ATN as it parses the following sentence:

The long file has printed.

This execution proceeds as follows:

1. Begin in state S.
2. Push to NP.
3. Do a category test to see if "the" is a determiner.
4. This test succeeds, so set the DETERMINER register to DEFINITE and go to state Q6.
5. Do a category test to see if "long" is an adjective.

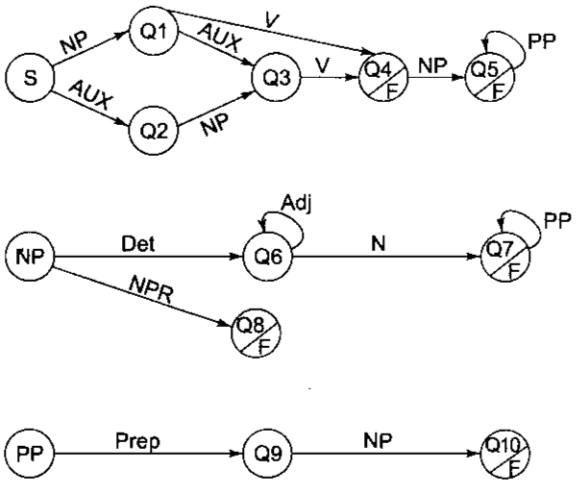


Fig. 15.8 *An ATN Network for a Fragment of English*

```
(S/     (PUSH NP/T
            (SETR SUBJ *)
            (SETR TYPE (QUOTE DCL))
            (TO Q1))
        (CAT AUX T
            (SETR AUX *)
            (SETR TYPE (OUOTE O))
            (TO Q2)))
(Ql     (CAT V T
            (SETR AUX NIL)
            (SETR V *)
            (TO O4))
        (CAT AUX T
            (SETR AUX *)
            (TO Q3)))
(Q2     (PUSH NP/ T
            (SETR SUBJ *)
            (TO Q3)))
(O3     (CAT V T
            (SETR V *)
            (TO Q4)))
(O4     (POP (BUILDQ (S + + + (VP +))
                TYPE SUBJ AUX V) T)
        (PUSH NP/ T
            (SETR VP (BUILDQ (VP (V +) *) V))
            (TO Q5)))
(Q5     (POP (BUILDQ (S + + + +)
                TYPE SUBJ AUX VP) T)
        (PUSH PP/ T
            (SETR VP (APPEND (GETR VP) (LIST *)))
            (TO Q5)))
```

Fig. 15.9 *An ATN Grammar in List Form*

6. This test succeeds, so append "long" to the list contained in the ADJS register. (This list was previously empty.) Stay in state Q6.
7. Do a category test to see if "file" is an adjective. This test fails.
8. Do a category test to see if "file" is a noun. This test succeeds, so set the NOUN register to "file" and go to state Q7.
9. Push to PP.
10. Do a category test to see if "has" is a preposition. This test fails, so pop and signal failure.
11. There is nothing else that can be done from state Q7, so pop and return the structure
    (NP (FILE (LONG) DEFINITE))
    The return causes the machine to be in state Q1, with the SUBJ register set to the structure just returned and the TYPE register set to DCL.
12. Do a category test to see if "has" is a verb. This test succeeds, so set the AUX register to NIL and set the V register to "has." Go to state Q4.
13. Push to state NP. Since the next word, "printed," is not a determiner or a proper noun, NP will pop and return failure.
14. The only other thing to do in state Q4 is to halt. But more input remains, so a complete parse has not been found. Backtracking is now required.
15. The last choice point was at state Q1, so return there. The registers AUX and V must be unset.
16. Do a category test to see if "has" is an auxiliary. This test succeeds, so set the AUX register to "has" and go to state Q3.
17. Do a category test to see if "printed" is a verb. This test succeeds, so set the V register to "printed." Go to state Q4.
18. Now, since the input is exhausted, Q4 is an acceptable final state. Pop and return the structure
    (S DCL (NP (FILE (LONG) DEFINITE)) HAS (VP PRINTED))
    This structure is the output of the parse.

This example grammar illustrates several interesting points about the use of ATNs. A single subnetwork need only occur once even though it is used in more than one place. A network can be called recursively. Any number of internal registers may be used to contain the result of the parse. The result of a network can be built, using the function BUILDQ, out of values contained in the various system registers. A single state may be both a final state, in which a complete sentence has been found, and an intermediate state, in which only a part of a sentence has been recognized. And, finally, the contents of a register can be modified at any time.

In addition, there are a variety of ways in which ATNs can be used which are not shown in this example:

- The contents of registers can be swapped. For example, if the network were expanded to recognize passive sentences, then at the point that the passive was detected, the current contents of the SUBJ register would be transferred to an OBJ register and the object of the preposition "by" would be placed in the SUBJ register. Thus the final interpretation of the following two sentences would be the same
  - Bill printed the file.
  - The file was printed by Bill.
- Arbitrary tests can be placed on the arcs. In each of the arcs in this example, the test is specified simply as T (always true). But this need not be the case. Suppose that when the first NP is found, its number is determined and recorded in a register called NUMBER. Then the arcs labeled V could have an additional test placed on them that checked that the number of the particular verb that was found is equal to the value stored in NUMBER. More sophisticated tests, involving semantic markers or other semantic features, can also be performed.

15.2.3 Unification Grammars

ATN grammars have substantial procedural components. The grammar describes the order in which constituents must be built. Variables are explicitly given values, and they must already have been assigned a value before they can be referenced. This procedurality limits the effectiveness of ATN grammars in some cases, for example: in speech processing where some later parts of the sentence may have been recognized clearly while earlier parts are still unknown (for example, suppose we had heard, "The long * * * file printed."), or in systems that want to use the same grammar to support both understanding and generation (e.g., Appelt [1987], Shieber [1988], and Barnett *et al.* [1990]). Although there is no clear distinction between declarative and procedural representations (as we saw in Section 6.1), there is a spectrum and it often turns out that more declarative representations are more flexible than more procedural ones are. So in this section we describe a declarative approach to representing grammars.

When a parser applies grammar rules to a sentence, it performs two major kinds of operations:

- Matching (of sentence constituents to grammar rules)
- Building structure (corresponding to the result of combining constituents)

Now think back to the unification operation that we described in Section 5.4.4 as part of our theorem-proving discussion. Matching and structure building are operations that unification performs naturally. So an obvious candidate for representing grammars is some structure on which we can define a unification operator. Directed acyclic graphs (DAGs) can do exactly that.

Each DAG represents a set of attribute-value pairs. For example, the graphs corresponding to the words "the" and "file" are:

```
[CAT: DET          [CAT: N
 LEX: the]          LEX: file
                    NUMBER: SING]
```

Both words have a lexical category (CAT) and a lexical entry. In addition, the word "file" has a value (SING) for the NUMBER attribute. The result of combining these two words to form a simple NP can also be described as a graph:

```
[NP: [DET: the
      HEAD: file
      NUMBER: SING]]
```

The rule that forms this new constituent can also be represented as a graph, but to do so we need to introduce a new notation. Until now, all our graphs have actually been trees. To describe graphs that are not trees, we need a way to label a piece of a graph and then point to that piece elsewhere in the graph. So let $\{n\}$ for any value of n be a label, which is to be interpreted as a label for the next constituent following it in the graph. Sometimes, the constituent is empty (i.e., there is not yet any structure that is known to fill that piece of the graph). In that case, the label functions very much like a variable and will be treated like one by the unification operation. It is this degenerate kind of a label that we need in order to describe the NP rule:

NP → DET N

We can write this rule as the following graph:

```
[CONSTITUENT1: [CAT: DET
                LEX: {1}]
CONSTITUENT2: [CAT: N
                LEX: {2}
                NUMBER: {3}]
BUILD: [NP:[DET: {1}
            HEAD: {2}
            NUMBER: {3}]]]
```

This rule should be read as follows: Two constituents, described in the subgraphs labeled CONSTITUENT1 and CONSTITUENT2, are to be combined. The first must be of CAT DET. We do not care what its lexical entry is, but whatever it is will be bound to the label {1}. The second constituent must be of CAT N. Its lexical entry will be bound to the label {2}, and its number will be bound to the label {3}. The result of combining these two constituents is described in the subgraph labeled BUILD. This result will be a graph corresponding to an NP with three attributes: DET, HEAD, and NUMBER. The values for all these attributes are to be taken from the appropriate pieces of the graphs that are being combined by the rule.

Now we need to define a unification operator that can be applied to the graphs we have just described. It will be very similar to logical unification. Two graphs unify if, recursively, all their subgraphs unify. The result of a successful unification is a graph that is composed of the union of the subgraphs of the two inputs, with all bindings made as indicated. This process bottoms out when a subgraph is not an attribute-value pair but is just a value for an attribute. At that point, we must define what it means for two values to unify. Identical values unify. Anything unifies with a variable (a label with no attached structure) and produces a binding for the label. The simplest thing to do is then to say that any other situation results in failure. But it may be useful to be more flexible. So some systems allow a value to match with a more general one (e.g., PROPER-NOUN matches NOUN). Others allow values that are disjunctions [e.g., (MASCULINE $\vee$ FEMININE)], in which case unification succeeds whenever the intersection of the two values is not empty.

There is one other important difference between logical unification and graph unification. The inputs to logical unification are treated as logical formulas. Order matters, since, for example, $f(g(a), h(b))$ is a different formula than $f(h(b), g(a))$. The inputs to graph unification, on the other hand, must be treated as sets, since the order in which attribute-value pairs are stated does not matter. For example, if a rule describes a constituent as

```
[CAT: DET
LEX: {1}]
```

we want to be able to match a constituent such as

```
[LEX: the
CAT: DET]
```

Algorithm: Graph-Unify

1. If either G_1 or G_2 is an attribute that is not itself an attribute-value pair then:
   (a) If the attributes conflict (as defined above), then fail.
   (b) If either is a variable, then bind it to the value of the other and return that value.
   (c) Otherwise, return the most general value that is consistent with both the original values. Specifically, if disjunction is allowed, then return the inter section of the values.

2. Otherwise, do:
   (a) Set variable *NEW* to empty.
   (b) For each attribute *A* that is present (at the top level) in either *G1* or *G2* do
      (i) If *A* is not present at the top level in the other input, then add *A* and its value to *NEW*.
      (ii) If it is, then call Graph-Unify with the two values for *A*. If that fails, then fail. Otherwise, take the new value of *A* to be the result of that unification and add *A* with its value to *NEW*.
   (c) If there are any labels attached to *G1* or *G2*, then bind them to *NEW* and return *NEW*.

A simple parser can use this algorithm to apply a grammar rule by unifying CONSTITUENT 1 with a proposed first constituent. If that succeeds, then CONSTITUENT2 is unified with a proposed second constituent. If that also succeeds, then a new constituent corresponding to the value of BUILD is produced. If there are variables in the value of BUILD that were bound during the matching of the constituents, then those bindings will be used to build the new constituent.

There are many possible variations on the notation we have described here. There are also a variety of ways of using it to represent dictionary entries and grammar rules. See Shieber [1986] and Knight [1989] for discussions of some of them.

Although we have presented unification here as a technique for doing syntactic analysis, it has also been used as a basis for semantic interpretation. In fact, there are arguments for using it as a uniform representation for all phases of natural language understanding. There are also arguments against doing so, primarily involving system modularity, the noncompositionality of language in some respects (see Section 15.3.4), and the need to invoke substantial domain reasoning. We will not say any more about this here, but to see how this idea could work, see Allen [1989].

15.3 SEMANTIC ANALYSIS

Producing a syntactic parse of a sentence is only the first step toward understanding it. We must still produce a representation of the *meaning* of the sentence. Because understanding is a mapping process, we must first define the language into which we are trying to map. There is no single, definitive language in which all sentence meanings can be described. All of the knowledge representation systems that were described in Part II are candidates, and having selected one or more of them, we still need to define the vocabulary (i.e., the predicates, frames, or whatever) that will be used on top of the structure. In the rest of this chapter, we call the final meaning representation language, including both the representational framework and the specific meaning vocabulary, the *target language*. The choice of a target language for any particular natural language understanding program must depend on what is to be done with the meanings once they are constructed. There are two broad families of target languages that are used in NL systems, depending on the role that the natural language system is playing in a larger system (if any).

When natural language is being considered as a phenomenon on its own, as, for example, when one builds a program whose goal is to read text and then answer questions about it, a target language can be designed specifically to support language processing. In this case, one typically looks for primitives that correspond to distinctions that are usually made in language. Of course, selecting the right set of primitives is not easy. We discussed this issue briefly in Section 4.3.3, and in Chapter 10 we looked at two proposals for a set of primitives, conceptual dependency and CYC.

When natural language is being used as an interface language to another program (such as a database query system or an expert system), then the target language must be a legal input to that other program. Thus the design of the target language is driven by the backend program. This was the case in the simple example we discussed in Section 15.1.1. But even in this case, it is useful, as we showed in that example, to use an intermediate knowledge-based representation to guide the overall process. So, in the rest of this section, we assume that the target language we are building is a knowledge-based one.

Although the main purpose of semantic processing is the creation of a target language representation of a sentence's meaning, there is another important role that it plays. It imposes constraints on the representations that can be constructed, and, because of the structural connections that must exist between the syntactic structure and the semantic one, it also provides a way of selecting among competing syntactic analyses. Semantic processing can impose constraints because it has access to knowledge about what makes sense in the world. We already mentioned one example of this, the sentence, *Is the glass jar peanut butter?" There are other examples in the rest of this section.

Lexical Processing

The first step in any semantic processing system is to look up the individual words in a dictionary (or *lexicon)* and extract their meanings. Unfortunately, many words have several meanings, and it may not be possible to choose the correct one just by looking at the word itself. For example, the word "diamond" might have the following set of meanings:

- A geometrical shape with four equal sides
- A baseball field
- An extremely hard and valuable gemstone

To select the correct meaning for the word "diamond" in the sentence,

> Joan saw Susan's diamond shimmering from across the room.

it is necessary to know that neither geometrical shapes nor baseball fields shimmer, whereas gemstones do.

Unfortunately, if we view English understanding as mapping from English words into objects in a specific knowledge base, lexical ambiguity is often greater than it seems in everyday English. For, example, consider the word "mean." This word is ambiguous in at least three ways: it can be a verb meaning "to signify"; it can be an adjective meaning "unpleasant" or "cheap"; and it can be a noun meaning "statistical average." But now imagine that we have a knowledge base that describes a statistics program and its operation. There might be at least two distinct objects in that knowledge base, both of which correspond to the "statistical average" meaning of "mean." One object is the statistical concept of a mean; the other is the particular function that computes the mean in this program. To understand the word "mean" we need to map it into some concept in our knowledge base. But to do that, we must decide which of these concepts is meant. Because of cases like this, lexical ambiguity is a serious problem, even when the domain of discourse is severely constrained.

The process of determining the correct meaning of an individual word is called *word sense disambiguation* or *lexical disambiguation.* It is done by associating, with each word in the lexicon, information about the contexts in which each of the word's senses may appear. Each of the words in a sentence can serve as part of the context in which the meanings of the other words must be determined.

Sometimes only very straightforward information about each word sense is necessary. For example, the baseball field interpretation of "diamond" could be marked as a LOCATION. Then the correct meaning of "diamond" in the sentence "I'll meet you at the diamond" could easily be determined if the fact that *at* requires a TIME or a LOCATION as its object were recorded as part of the lexical entry for *at.* Such simple properties of word senses are called *semantic markers.* Other useful semantic markers are

- PHYSICAL-OBJECT
- ANIMATE-OBJECT
- ABSTRACT-OBJECT

Using these markers, the correct meaning of "diamond" in the sentence "I dropped my diamond" can be computed. As part of its lexical entry, the verb "drop" will specify that its object must be a PHYSICAL-

OBJECT. The gemstone meaning of "diamond" will be marked as a PHYSICAL-OBJECT. So it will be selected as the appropriate meaning in this context.

This technique has been extended by Wilks [1972; 1975a; 1975b] in his *preference semantics*, which relies on the notion that requirements, such as the one described above for an object that is a LOCATION, are rarely hard-and-fast demands. Rather, they can best be described as preferences. For example, we might say that verbs such as "hate" prefer a subject that is animate. Thus we have no difficulty in understanding the sentence

Pop hates the cold.

as describing the feelings of a man and not those of soft drinks. But now consider the sentence

My lawn hates the cold.

Now, there is no animate subject available, and so the metaphorical use of lawn acting as an animate object should be accepted.

Unfortunately, to solve the lexical disambiguation problem completely, it becomes necessary-to introduce more and more finely grained semantic markers. For example, to interpret the sentence about Susan's diamond correctly, we must mark one sense of diamond as SHIMMERABLE, while the other two are marked NONSHIMMERABLE. As the number of such markers grows, the size of the lexicon becomes unmanageable. In addition, each new entry into the lexicon may require that a new marker be added to each of the existing entries. The breakdown of the semantic marker approach when the number of words and word senses becomes large has led to the development of other ways in which correct senses can be chosen. We return to this issue in Section 15.3.4.

Sentence-Level Processing

Several approaches to the problem of creating a semantic representation of a sentence have been developed, including the following:

- Semantic grammars, which combine syntactic, semantic, and pragmatic knowledge into a single set of rules in the form of a grammar. The result of parsing with such a grammar is a semantic, rather than just a syntactic, description of a sentence.
- Case grammars, in which the structure that is built by the parser contains some semantic information, although further interpretation may also be necessary.
- Conceptual parsing, in which syntactic and semantic knowledge are combined into a single interpretation system that is driven by the semantic knowledge. In this approach, syntactic parsing is subordinated to semantic interpretation, which is usually used to set up strong expectations for particular sentence structures,
- Approximately compositional semantic interpretation, in which semantic processing is applied to the result of performing a syntactic parse. This can be done either incrementally, as constituents are built, or all at once, when a structure corresponding to a complete sentence has been built.

In the following sections, we discuss each of these approaches.

15.3.1 Semantic Grammars

A *semantic grammar* [Burton; 1976; Hendrix *et al.*, 1978; Hendrix and Lewis, 1981] is a context-free grammar in which the choice of nonterminals and production rules is governed by semantic as well as syntactic function. In addition, there is usually a semantic action associated with each grammar rule. The result of parsing and applying all the associated semantic actions is the meaning of the sentence. This close coupling of semantic

actions to grammar rules works because the grammar rules themselves are designed around key semantic concepts.

An example of a fragment of a semantic grammar is shown in Fig. 15.10. This grammar defines part of a simple interface to an operating System. Shown in braces under each rule is the semantic action that is taken when the rule is applied. The term "value" is used to refer to the value that is matched by the right-hand side of the rule. The dotted notation *x.y* should be read as the *y* attribute of the unit *x*. The result of a successful parse using this grammar will be either a command or a query.

```
S → what is FILE-PROPERTY of FILE?
        {query FILE.FILE-PROPERTY}
SK → I want to ACTION
        {command ACTION}
FILE-PROPERTY → the FILE-PROP
        {FILE-PROP}
FILE-PROP → extension I protection I creation date I owner
        {value}
FILE → FILE-NAME I FILE1
        {value}
FILE1 → USER's FILE2
        {FILE2.owner: USER}
FILE1 → FILE2
        {FILE2}
FILE2 → EXT file
        {instance: file-struct
        extension: EXT}
EXT → .init I .txt I .lsp I .for I .ps I .mss
        value
ACTION → print FILE
        {instance: printing
        object: FILE}
ACTION → print FILE on PRINTER
        {instance: printing
        object: FILE
        printer: PRINTER}
USER → Bill I Susan
        {value}
```

Fig. 15.10 *A Semantic Grammar*

A semantic grammar can be used by a parsing system in exactly the same ways in which a strictly syntactic grammar could be used. Several existing systems that have used semantic grammars have been built around an ATN parsing system, since it offers, a great deal of flexibility.

Figure 15.11 shows the result of applying this semantic grammar to the sentence

I want to print Bill's .init file.

Notice that in this approach, we have combined into a single process all five steps of Section 15.1.1 with the exception of the final part of pragmatic processing in which the conversion to the system's command syntax is done.

The principal advantages of semantic grammars are the following:

- When the parse is complete, the result can be used immediately without the additional stage of processing that would be required if a semantic interpretation had not already been performed during the parse.

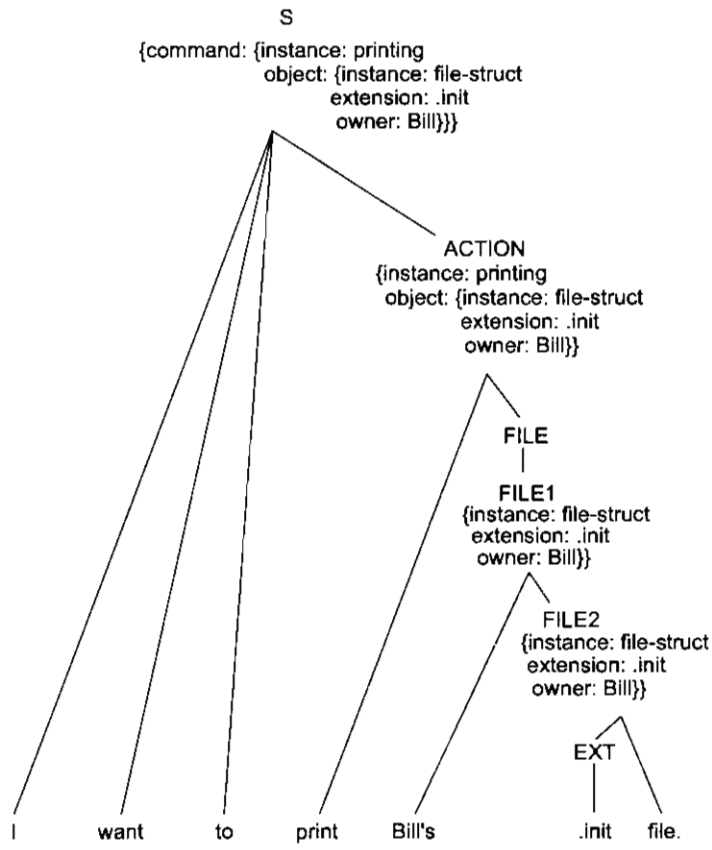


Fig. 15.11 *The Result of Parsing with a Semantic Grammar*

- Many ambiguities that would arise during a strictly syntactic parse can be avoided since some of the interpretations do not make sense semantically and thus cannot be generated by a semantic grammar. Consider, for example, the sentence "I want to print stuff.txt on printer3." During a strictly syntactic parse, it would not be possible to decide whether the prepositional phrase, "on printer3" modified "want" or "print." But using our semantic grammar, there is no general notion of a prepositional phrase and there is no attachment ambiguity.
- Syntactic issues that do not affect the semantics can be ignored. For example, using the grammar shown above, the sentence, "What is the extension of .lisp file?" would be parsed and accepted as correct.

There are, however, some drawbacks to the use of semantic grammars:

- The number of rules required can become very large since many syntactic generalizations are missed.
- Because the number of grammar rules may be very large, the parsing process may be expensive.

After many experiments with the use of semantic grammars in a variety of domains, the conclusion appears to be that for producing restricted natural language interfaces quickly, they can be very useful. But as an overall solution to the problem of language understanding, they are doomed by their failure to capture important linguistic generalizations.

15.3.2 Case Grammars

Case grammars [Fillmore, 1968; Bruce, 1975] provide a different approach to the problem of how syntactic and semantic interpretation can be combined. Grammar rules are written to describe syntactic rather than semantic regularities. But the structures the rules produce correspond to semantic relations rather than to strictly syntactic ones. As an example, consider the two sentences and the simplified forms of their conventional parse trees shown in Fig. 15.12.

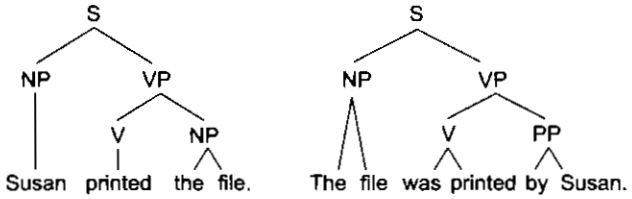


Fig. 15.12 *Syntactic Parses of an Active and a Passive Sentence*

Although the semantic roles of "Susan" and "the file" are identical in these two sentences, their syntactic roles are reversed. Each is the subject in one sentence and the object in another.

Using a case grammar, the interpretations of the two sentences would both be

```
(printed (agent Susan)
         (object File))
```

Now consider the two sentences shown in Fig. 15.13.

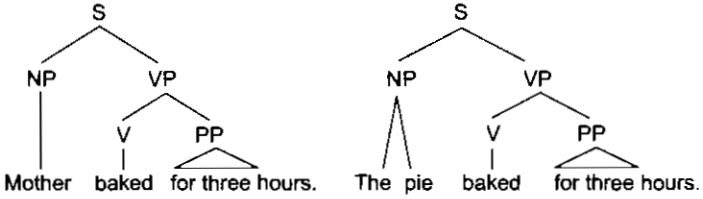


Fig. 15.13 *Syntactic Parses of Two Similar Sentences*

The syntactic structures of these two sentences are almost identical. In one case, "Mother" is the subject of "baked," while in the other "the pie" is the subject. But the relationship between Mother and baking is very different from that between the pie and baking. A case grammar analysis of these two sentences reflects this difference. The first sentence would be interpreted as

```
(baked (agent Mother)
       (timeperiod 3-hours))
```

The second would be interpreted as

```
(baked (object Pie)
       (timeperiod 3-hours))
```

In these representations, the semantic roles of "mother" and "the pie" are made explicit. It is interesting to note that this semantic information actually does intrude into the syntax of the language. While it is allowed to conjoin two parallel sentences (e.g., "the pie baked" and "the cake baked" become "the pie and the cake

baked"), this is only possible if the conjoined noun phrases are in the same case relation to the verb. This accounts for the fact that we do not say, "Mother and the pie baked."

Notice that the cases used by a case grammar describe relationships between verbs and their arguments. This contrasts with the grammatical notion of surface case, as exhibited, for example, in English, by the distinction between "I" (nominative case) and "me" (objective case). A given grammatical, or surface, case can indicate a variety of semantic, or deep, cases.

There is no clear agreement on exactly what the Correct set of deep cases ought to be, but some obvious ones are the following:

- (A) Agent—Instigator of the action (typically animate)
- (I) Instrument—Cause of the event or object used in causing the event (typically inanimate)
- (D) Dative—Entity affected by the action (typically animate)
- (F) Factitive—Object or being resulting from the event
- (L) Locative—Place of the event
- (S) Source—Place from which something moves
- (G) Goal—Place to which something moves
- (B) Beneficiary—Being on whose behalf the event occurred (typically animate)
- (T) Time—Time at which the event occurred
- (O) Object—Entity that is acted upon or that changes, the most general case

The process of parsing into a case representation is Heavily directed by the lexical entries associated with each verb. Figure 15.14 shows examples of a few such entries. Optional cases are indicated in parentheses.

open	[_ _ O (I) (A)]
	The door opened.
	John opened the door.
	The wind opened the door.
	John opened the door with a chisel.
die	[_ _ D]
	John died.
kill	[_ _ D (I) A]
	Bill killed John.
	Bill killed John with a knife.
run	[_ _ A]
	John ran.
want	[_ _ A O]
	John wanted some ice cream.
	John wanted Mary to go to the store.

Fig. 15.14 *Some Verb Case Frames*

Languages have rules for mapping from underlying case structures to surface syntactic forms. For example, in English, the "unmarked subject"[3] is generally chosen hy the following rule:

> If A is present, it is the subject. Otherwise, if I is present, it is the subject. Else the subject is O.

[3]The unmarked subject is the one that is used by default; it signals no special focus or emphasis in the sentence.

The [illegible] superficial syntax.

Parsing using a case grammar is usually *expectation-driven*. Once the verb of the sentence has been located, it can be used to predict the noun phrases that will occur and to determine the relationship of those phrases to the rest of the sentence.

ATNs provide a good structure for case grammar parsing. Unlike traditional parsing algorithms in which the output structure always mirrors the structure of the grammar rules that created it, ATNs allow output structures of arbitrary form. For an example of their use, see Simmons [1973], which describes a system that uses an ATN parser to translate English sentences into a semantic net representing the case structures of sentences. These semantic nets can then be used to answer questions about the sentences.

The result of parsing in a case representation is usually not a complete semantic description of a sentence. For example, the constituents that fill the case slots may still be English words rather than true semantic descriptions stated in the target representation. To go the rest of the way toward building a meaning representation, we still require many of the steps that are described in Section 15.3.4.

15.3.3 Conceptual Parsing

Conceptual parsing, like semantic grammars, is a strategy for finding both the structure and the meaning of a sentence in one step. Conceptual parsing is driven by a dictionary that describes the meanings of words as conceptual dependency (CD) structures.

Parsing a sentence into a conceptual dependency representation is similar to the process of parsing using a case grammar. In both systems, the parsing process is heavily driven by a set of expectations that are set up on the basis of the sentence's main verb. But because the representation of a verb in CD is at a lower level than that of a verb in a case grammar (in which the representation is often identical to the English word that is used), CD usually provides a greater degree of predictive power. The first step in mapping a sentence into its CD representation involves a syntactic processor that extracts the main noun and verb. It also determines the syntactic category and aspectual class of the verb (i.e., stative, transitive, or intransitive). The conceptual processor then takes over. It makes use of a verb-ACT dictionary, which contains an entry for each environment in which a verb can appear. Figure 15.15 (taken from Schank [1973]) shows the dictionary entries associated with the verb "want." These three entries correspond to the three kinds of wanting:

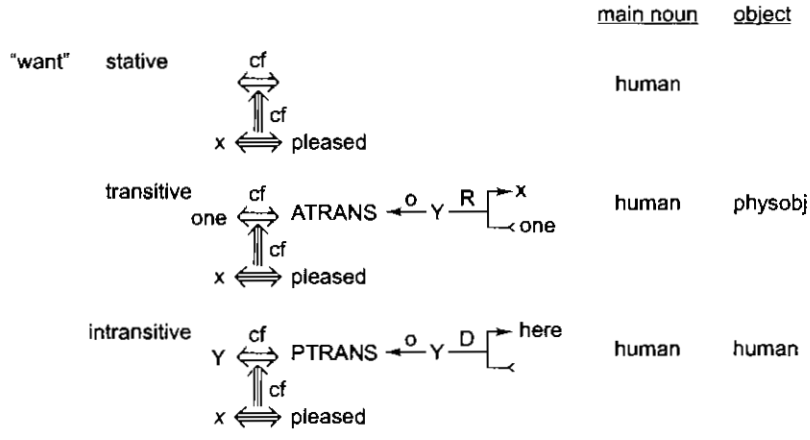


Fig. 15.15 *The Verb-ACT Dictionary*

- Wanting something to happen
- Wanting an object
- Wanting a person

Once the correct dictionary entry is chosen, the conceptual processor analyzes the rest of the sentence looking for components that will fit into the empty slots of the verb structure. For example, if the stative form of "want" has been found, then the conceptual processor will look for a conceptualization that can be inserted into the structure. So, if the sentence being processed were

John wanted Mary to go to the store.

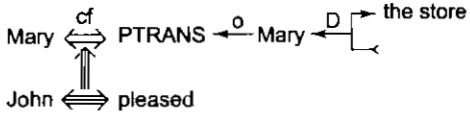


Fig. 15.16 *A CD Structure*

the structure shown in Fig. 15.16 would be built.

The conceptual processor examines possible interpretations in a well-defined order. For example, if a phrase of the form "'with PP" (recall that a PP is a picture producer) occurs, it could indicate any of the following relationships between the PP and the conceptualization of which it is a part:

1. Object of the instrumental case
2. Additional actor of the main ACT
3. Attribute of the PP just preceding it
4. Attribute of the actor of the conceptualization

Suppose that the conceptual processor were attempting to interpret the prepositional phrase in the sentence

John went to the park with the girl.

First, the system's immediate memory would be checked to see if a park with a girl has been mentioned. If so, a reference to that particular object is generated and the process terminates. Otherwise, the four possibilities outlined above are investigated in the order in which they are presented. Can "the girl" be an instrument of the main ACT (PTRANS) of this sentence? The answer is no, because only MOVE and PROPEL can be instruments of a PTRANS and their objects must be either body parts or vehicles. "Girl" is neither of these. So we move on to consider the second possibility. In order for "girl" to be an additional actor of the main ACT, it must be animate. It is. So this interpretation is chosen and the process terminates. If, however, the sentence had been

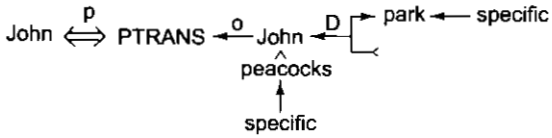
John went to the park with the fountain.

the process would not have stopped since a fountain is inanimate and cannot move. Then the third possibility would have been considered. Since parks can have fountains, it would be accepted and the process would terminate there. For a more detailed description of the way a conceptual processor based on CD works, see Schank [1973], Rieger [1975], and Riesbeck [1975].

This example illustrates both the strengths and the weaknesses of this approach to sentence understanding. Because a great deal of semantic information is exploited in the understanding process, sentences that would be ambiguous to a purely syntactic parser can be assigned a unique interpretation. Unfortunately, the amount of semantic information that is required to do this job perfectly is immense. All simple rules have exceptions. For example, suppose the conceptual processor described above were given the sentence

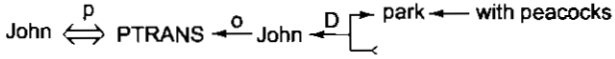
John went to the park with the peacocks.

the interpretation that would be produced would be that shown in Fig. 15.17(a), while the more likely interpretation, in which John went to a park containing peacocks, is shown in Fig. 15.17(b). But if the possible roles for a prepositional phrase introduced by "with" were considered in the order necessary for this sentence to be interpreted correctly, then the previous example involving the phrase, "with Mary," would have been misunderstood.



John went to the park with the peacocks.

(a)



John went to the park with the peacocks.

(b)

Fig. 15.17 *Two CD Interpretations of a Sentence*

The problem is that the simple check for the property ANIMATE is not sufficient to determine acceptability as an additional actor of a PTRANS. Additional knowledge is necessary. Some more knowledge can be inserted within the framework we have described for a conceptual processor. But to do a very good job of producing correct semantic interpretations of sentences requires knowledge of the larger context in which the sentence appears. Techniques for exploiting such knowledge are discussed in the next section.

15.3.4 Approximately Compositional Semantic Interpretation

The final approach to semantics that we consider here is one in which syntactic parsing and semantic interpretation are treated as separate steps, although they must mirror each other in well-defined ways. This is the approach to semantics that we looked at briefly in Section 15.1.1 when we worked through the example sentence "I want to print Bill's .init file."

If a strictly syntactic parse of a sentence has been produced then a straightforward way to generate a semantic interpretation is the following:

1. Look up each word in a lexicon that contains one or more definitions for the word, each stated in terms of the chosen target representation. These definitions must describe how the idea that corresponds to the word is to be represented, and they may also describe how the idea represented by this word may combine with the ideas represented by other words in the sentence.
2. Use the structure information contained in the output of the parser to provide additional constraints, beyond those extracted from the lexicon, on the way individual words may combine to form larger meaning units.

We have already discussed the first of these steps (in Section 15.3). In the rest of this section, we discuss the second.

Montague Semantics

Recall that we argued in Section 15.1.1 that the reason syntactic parsing was a good idea was that it produces structures that correspond to the structures that should result from semantic processing. If we investigate this

idea more closely, we arrive at a notion called *compositional semantics*. The main idea behind compositional semantics is that, for every step in the syntactic parsing process, there is a corresponding step in semantic interpretation. Each time syntactic constituents are combined to form a larger syntactic unit, their corresponding semantic interpretations can be combined to form a larger semantic unit. The necessary rules for combining semantic structures are associated with the corresponding rules for combining syntactic structures. We use the word "compositional" to describe this approach because it defines the meaning of each sentence constituent to be a composition of the meanings of its constituents with the meaning of the rule that was used to create it. The main theoretical basis for this approach is modern (i.e., post-Fregean) logic; the clearest linguistic application is the work of Montague [Dowty *et al.*, 1981; Thomason, 1974].

As an example of this approach to semantic interpretation, let's return to the example that we began in Section 15.1.1. The sentence is

I want to print Bill's .init file.

The output of the syntactic parsing process was shown in Fig. 15.2, and a fragment of the knowledge base that is being used to define the target representation was shown in Fig. 15.3. The result of semantic interpretation was also shown there in Fig. 15.4. Although the exact form of semantic mapping rules in this approach depends on the way that the syntactic grammar is defined, we illustrate the idea of compositional semantic rules in Fig. 15.18.

"want" *subject:* RM_i *object:* RM_j	→	*Unit* *instance: Wanting* *agent:* RM_i *object:* RM_j
"print" *subject:* RM_i *object:* RM_j	→	*Unit* *instance: Printing* *agent:* RM_i *object:* RM_j
".init" modifying NP_1	→	*Unit* for NP_1 plus *extension:* .init
possessive marker NP_1's NP_2	→	*Unit* for NP_2 plus *owner:* NP_1
"file"	→	*Unit* *instance: File-Struct*
"Bill"	→	*Unit* *instance: Person* *first-name:* Bill

Fig. 15.18 *Some Semantic Interpretation Rules*

The first two rules are examples of verb-mapping rules. Read these rules as saying that they map from a partial syntactic structure containing a verb, its subject, and its object, to some unit with the attributes instance, agent, and object. These rules do two things. They describe the meaning of the verbs ("want" or "print") themselves in terms of events in the knowledge base. They also state how the syntactic arguments of the verbs (their subjects and objects) map into attributes of those events. By the way, do not get confused by the use of the term "object" in two different senses here. The syntactic object of a sentence and its semantic object are two different things. For historical reasons (including the standard usage in case grammars as described in Section 15.3.2), they are often called the same thing, although this problem is sometimes avoided by using some other name, such as *affected-entity*, for the semantic object. Alternatively, in some knowledge bases, much more specialized names, such as *printed-thing*, are sometimes used as attribute names.

their own constituent's contribution to meaning as well as how it combines with the meaning of the noun phrase or phrases to which it is attached.

The last two rules are simpler. They define the meanings of nouns. Since nouns do not usually take arguments, these rules specify only single-word meanings; they do not need to describe how the meanings of larger constituents are derived from their components.

One important thing to remember about these rules is that since they define mappings from words into a knowledge base, they implicitly make available to the semantic processing system all the information contained in the knowledge base itself. For example, Fig. 15.19 contains a description of the semantic information that is associated with the word "want" after applying the semantic rule associated with the verb" and retrieving semantic constraints associated with wanting events in the knowledge base. Notice that we now know where to pick up the agent for the wanting (*RM*1) and we now know some property that the agent must have. The semantic interpretation routine will reject any interpretation that does not satisfy all these constraints.

This compositional approach to defining semantic interpretation has proved to be a very powerful idea. (See, for example, the Absity system described in Hirst [1987].) Unfortunately, there are some linguistic constructions that cannot be accounted for naturally in a strictly compositional system. Quantified expressions have this property. Consider, for example, the sentence

Every student who hadn't declared a major took an English class.

Unit
instance: *Wanting*
agent: RM_i
must be <animate>
object: RM_j
must be <state or event>

Fig. 15.19 *Combining Mapping Knowledge with the Knowledge Base*

There are several ways in which the relative scopes of the quantifiers in this sentence can be assigned. In the most likely, both existential quantifiers are within the scope of the universal quantifier. But, in other readings, they are not. These include readings corresponding to, "There is a major such that every student who had not declared it took an English class," and "There is an English class such that every student who had not declared some major took it." In order to generate these meanings compositionally from the parse, it is necessary to produce a separate parse for each scope assignment. But there is no syntactic reason to do that, and it requires substantial additional effort. An alternative is to generate a single parse and then to use a noncompositional algorithm to generate as many alternative scopes as desired.

As a second example, consider the sentence, "John only eats meat on Friday and Mary does too." The syntactic analysis of this sentence must include the verb phrase constituent, "only eats meat on Friday," since that is the constituent that is picked up by the elliptical expression "does too." But the meaning of the first clause has a structure more like

only(meat, {x | John eats x on Friday})

which can be read as, "Meat is the only thing that John eats on Friday."

Extended Reasoning with a Knowledge Base

A significant amount of world knowledge may be necessary in order to do semantic interpretation (and thus, sometimes, to get the correct syntactic parse). Sometimes the knowledge is needed to enable the system to choose among competing interpretations. Consider, for example, the sentences

1. John made a huge wedding cake with chocolate icing.
2. John made a huge wedding cake with Bill's mixer.
3. John made a huge wedding cake with a giant tower covered with roses.
4. John made a cherry pie with a giant tower covered with roses.

Let us concentrate on the problem of deciding to which constituent the prepositional phrase should be attached and of assigning a meaning to the preposition "with." We have two main choices: either the phrase attaches to the action of making the cake and "with" indicates the instrument relation, or the prepositional phrase attaches to the noun phrase describing the dessert that was made, in which case "with" describes an additional component of the dessert. The first two sentences are relatively straightforward if we imagine that our knowledge base contains the following facts:

- Foods can be components of other foods.
- Mixers are used to make many kinds of desserts.

But now consider the third sentence. A giant tower is neither a food nor a mixer. So it is not a likely candidate for either role. What is required here is the much more specific (and culturally dependent) fact that

- Wedding cakes often have towers and statues and bridges and flowers on them.

The highly specific nature of this knowledge is illustrated by the fact that the last of these sentences does not make much sense to us since we can find no appropriate role for the tower, either as part of a pie or as an instrument used during pie making.

Another use for knowledge is to enable the system to accept meanings that it has not been explicitly told about. Consider the following sentences as examples:

1. Sue likes to read Joyce.
2. Washington backed out of the summit talks.
3. The stranded explorer ate squirrels.

Suppose our system has only the following meanings for the words "Joyce," "Washington," and "squirrel" (actually we give only the relevant parts of the meanings):

1. Joyce—*instance: Author; last-name:* Joyce
2. Washington—*instance. City*; *name:* Washington
3. squirrel—*isa: Rodent;...*

But suppose that we also have only the following meanings for the verbs in these sentences:

1. read—*isa: Mental-Event; object:* must be <printed-material>
2. back out—*isa: Mental-Event; agent:* must be <animate-entity>
3. eat—*isa: Ingestion-Event*; *object:* must be <food>

The problem is that it is not possible to construct coherent interpretations for any of these sentences with these definitions. An author is not a <printed-material>. A city is not an <animate-entity>. A rodent is not a <food>. One solution is to create additional dictionary entries for the nouns: Joyce as a set of literary works, Washington as the people who run the U.S. government, and a squirrel as a food. But a better solution is to use general knowledge to derive these meanings when they are needed. By better, here we mean that since less knowledge must be entered by hand, the resulting system will be less brittle. The general knowledge that is necessary to handle these examples is:

- The name of a place can be used to stand for an organization headquartered in that place if the association between the organization and the place is salient in the context. An organization can in turn stand for the people who run it. The headquarters of the U.S. government is in Washington.
- Food (meat) can be made out of almost any animal. Usually the word for the animal can be used to refer to the meat made from the animal.

Of course, this problem can become arbitrarily complex. For example, metaphors are a rich source for linguistic expressions [Lakoff and Johnson, 1980]. And the problem becomes even more complex when we move beyond single sentences and attempt to extract meaning from texts and dialogues. We delve briefly into those issues in Section 15.4.

The Interaction between Syntax and Semantics

If we take a compositional approach to semantics, then we apply semantic interpretation rules to each syntactic constituent, eventually producing an interpretation for an entire sentence. But making a commitment about what to do implies no specific commitment about when to do it. To implement a system, however, we must make some decision on how control will be passed back and forth between the syntactic and the semantic processors. Two extreme positions are:

- Every time a syntactic constituent is formed, apply semantic interpretation to it immediately.
- Wait until the entire sentence has been parsed, and then interpret the whole thing.

There are arguments in favor of each approach. The theme of most of the arguments is search control and the opportunity to prune dead-end paths. Applying semantic processing to each constituent as soon as it is produced allows semantics to rule out right away those constituents that are syntactically valid but that make no sense. Syntactic processing can then be informed that it should not go any further with those constituents. This approach would pay off, for example, for the sentence, "Is the glass jar peanut butter?" But this approach can be costly when syntactic processing builds constituents that it will eventually reject as being syntactically unacceptable, regardless of their semantic acceptability. The sentence, "The horse raced past the barn fell down," is an example of this. There is no point in doing a semantic analysis of the sentence "The horse raced past the barn," since that constituent will not end up being part of any complete syntactic parse. There are also additional arguments for waiting until a complete sentence has been parsed to do at least some parts of semantic interpretation. These arguments involve the need for large constituents to serve as the basis of those semantic actions, such as the ones we discussed in Section 15.3.4, that are hard to define completely compositionally. There is no magic solution to this problem. Most systems use one of these two extremes or a heuristically driven compromise position.

15.4 DISCOURSE AND PRAGMATIC PROCESSING

To understand even a single sentence, it is necessary to consider the discourse and pragmatic context in which the sentence was uttered (as we saw in Section 15.1.1). These issues become even more important when we want to understand texts and dialogues, so in this section we broaden our concern to these larger linguistic units. There are a number of important relationships that may hold between phrases and parts of their discourse contexts, including:

- Identical entities. Consider the text
  - Bill had a red balloon.
  - John wanted it.

The word "it" should be identified as referring to the red balloon. References such as this are called *anaphoric references* or *anaphora*.

- Parts of entities. Consider the text
  - Sue opened the book she just bought.
  - The title page was torn.

  The phrase "the title page" should be recognized as being part of the book that was just bought.

- Parts of actions. Consider the text
  - John went on a business trip to New York.
  - He left on an early morning flight.

  Taking a flight should be recognized as part of going on a trip.

- Entities involved in actions. Consider the text
  - My house was broken into last week.
  - They took the TV and the stereo.

  The pronoun "they" should be recognized as referring to the burglars who broke into the house.

- Elements of sets. Consider the text
  - The decals we have in stock are stars, the moon, item and a flag.
  - I'll take two moons.

  The moons in the second sentence should be understood to be some of the moons mentioned in the first sentence. Notice that to understand the second sentence at all requires that we use the context of the first sentence to establish that the word "moons" means moon decals.

- Names of individuals. Consider the text
  - Dave went to the movies.

  Dave should be understood to be some person named Dave. Although there are many, the speaker had one particular one in mind and the discourse context should tell us which.

- Causal chains. Consider the text
  - There was a big snow storm yesterday.
  - The schools were closed today.

  The snow should be recognized as the reason that the schools were closed.

- Planning sequences. Consider the text
  - Sally wanted a new car.
  - She decided to get a job.

  Sally's sudden interest in a job should be recognized as arising out of her desire for a new car and thus for the money to buy one.

- Illocutionary force. Consider the sentence
  - It sure is cold in here.

In many circumstances, this sentence should be recognized as having, as its intended effect, that the hearer should do something like close the window or turn up the thermostat.

- Implicit presuppositions. Consider the query
  - Did Joe fail CS101?

  The speaker's presuppositions, including the fact that CS 101 is a valid course, that Joe is a student, and that Joe took CS 101, should be recognized so that if any of them is not satisfied, the speaker can be informed.

In order to be able to recognize these kinds of relationships among sentences, a great deal of knowledge about the world being discussed is required. Programs that can do multiple-sentence understanding rely either on large knowledge bases or on strong constraints on the domain of discourse so that only a more limited knowledge base is necessary. The way this knowledge is organized is critical to the success of the understanding program. In the rest of this section, we discuss briefly how some of the knowledge representations described in Chapters 9 and 10 can be exploited by a language-understanding program. In particular, we focus on the use of the following kinds of knowledge:

- The current focus of the dialogue
- A model of each participant's current beliefs
- The goal-driven character of dialogue
- The rules of conversation shared by all participants

Although these issues are complex, we discuss them only briefly here. Most of the hard problems are not peculiar to natural language processing. They involve reasoning about objects, events, goals, plans, intentions, beliefs, and likelihoods, and we have discussed all these issues in some detail elsewhere. Our goal in this section is to tie those reasoning mechanisms into the process of natural language understanding.

15.4.1 Using Focus in Understanding

There are two important parts of the process of using knowledge to facilitate understanding:

- Focus on the relevant part(s) of the available knowledge base.
- Use that knowledge to resolve ambiguities and to make connections among things that were said.

The first of these is critical if the amount of knowledge available is large. Some techniques for handling this were outlined in Section 4.3.5, since the problem arises whenever knowledge structures are to be used.

The linguistic properties of coherent discourse, however, provide some additional mechanisms for focusing. For example, the structure of task-oriented discourses typically mirrors the structure of the task. Consider the following sequence of (highly simplified) instructions:

> To make the torte, first make the cake, then, while the cake is baking, make the filling. To make the cake, combine all ingredients. Pour them into the pans, and bake for 30 minutes. To make the filling, combine the ingredients. Mix until light and fluffy. When the cake is done, alternate layers of cake and filling.

This task decomposes into three subtasks: making the cake, making the filling, and combining the two components. The structure of the paragraph of instructions is: overall sketch of the task, instructions for step 1, instructions for step 2, and then instructions for step 3.

A second property of coherent discourse is that dramatic changes of focus are usually signaled explicitly with phrases such as "on the other hand," "to return to an earlier topic," or "a second issue is."

Assuming that all this knowledge has been used successfully to focus on the relevant part(s) of the knowledge base, the second issue is how to use the focused knowledge to help in understanding. There are as many ways of doing this as there are discourse phenomena that require it. In the last section, we presented a sample list of those phenomena. To give one example, consider the problem of finding the meaning of definite noun phrases. Definite noun phrases are ones that refer to specific individual objects, for example, the first noun phrase in the sentence, "The title page was torn." The title page in question is assumed to be one that is related to an object that is currently in focus. So the procedure for finding a meaning for it involves searching for ways in which a title page could be related to a focused object. Of course, in some sense, almost any object in a knowledge base relates somehow to almost any other. But some relations are far more salient than others, and they should be considered first. Highly salient relations include *physical-part-of*, *temporal-part-of*, and *element-of*. In this example, *physical-part-of* relates the title page to the book that is in focus as a result of its mention in the previous sentence.

Other ways of using focused information also exist. We examine some of them in the remaining parts of this section.

15.4.2 Modeling Beliefs

In order for a program to be able to participate intelligently in a dialogue, it must be able to represent not only its own beliefs about the world, but also its knowledge of the other dialogue participant's beliefs about the world, that person's beliefs about the computer's beliefs, and so forth. The remark "She knew I knew she knew I knew she knew"[4] may be a bit extreme, but we do that kind of thinking all the time. To make computational models of belief, it is useful to divide the issue into two parts: those beliefs that can be assumed to be shared among all the participants in a linguistic event and those that cannot.

Modeling Shared Beliefs

Shared beliefs can be modeled without any explicit notion of belief in the knowledge base. All we need to do is represent the shared beliefs as facts, and they will be accessed whenever knowledge about anyone's beliefs is needed. We have already discussed techniques for doing this. For example, much of the knowledge described in Chapter 10 is exactly the sort that people presume is shared by other people they are communicating with. Scripts, in particular, have been used extensively to aid in natural language understanding. Recall that scripts record commonly occurring sequences of events. There are two steps in the process of using a script to aid in language understanding:

- Select the appropriate script(s) from memory.
- Use the script(s) to fill in unspecified parts of the text to be understood.

Both of these aspects of reasoning with scripts have already been discussed in Section 10.2. The story-understanding program SAM [Cullingford, 1981] demonstrated the usefulness of such reasoning with scripts in natural language understanding. To understand a story, SAM first employed a parser that translated the English sentences . into their conceptual dependency representation. Then it built a representation of the entire text using the relationships indicated by the relevant scripts.

Modeling Individual Beliefs

As soon as we decide to represent individual beliefs, we need to introduce some explicit predicate(s) to indicate that a fact is believed. Up until now, belief has been indicated only by the presence or absence of assertions in the knowledge base. To model belief, we need to move to a logic that supports reasoning about

[4] From Kingsley Amis' *Jake's Thing*.

belief propositions. The standard approach is to use a *modal logic* such as that defined in Hintikka [1962]. Logic, or "classical" logic, deals with the truth or falsehood of different statements as they are. Modal logic, on the other hand, concerns itself with the different "modes" in which a statement may be true. Modal logics allow us to talk about the truth of a set of propositions not only in the current state of the real world, but also about their truth or falsehood in the past or the future (these are called *temporal logics*), and about their truth or falsehood under circumstances that might have been, but were not (these are sometimes called *conditional logics*). We have already used one idea from modal logic, namely the notion *necessarily true*. We used it in Section 13.5, when we talked about nonlinear planning in TWEAK.

Modal logics also allow us to talk of the truth or falsehood of statements concerning the beliefs, knowledge, desires, intentions, and obligations of people and robots, which may, in fact be, respectively, false, unjustified, unsatisfiable, irrational, or mutually contradictory. Modal logics thus provide a set of powerful tools for understanding natural language utterances, which often involve reference to other times and circumstances, and to the mental states of people.

In particular, to model individual belief we define a modal operator BELIEVE, that enables us to make assertions of the form BELIEVE(A, P), which is true whenever A believes P to be true. Notice that this can occur even if P is believed by someone else to be false or even if P is false.

Another useful modal operator is KNOW:

$$\text{BELIEVE}(A, P) \wedge P \rightarrow \text{KNOW}(A, P)$$

A third useful modal operator is KNOW-WHAT(A, P), which is true if A knows the value of the function P. For example, we might say that A knows the value of his age.

An alternative way to represent individual beliefs is to use the idea of knowledge base partitioning that we discussed in Section 9.1. Partitioning enables us to do two things:

1. Represent efficiently the large set of beliefs shared by the participants. We discussed one way of doing this above.
2. Represent accurately the smaller set of beliefs that are not shared.

Requirement 1 makes it imperative that shared beliefs not be duplicated in the representation. This suggests that a single knowledge base must be used to represent the beliefs of all the participants. But requirement 2 demands that it be possible to separate the beliefs of one person from those of another. One way to do this is to use partitioned semantic nets. Figure 15.20 shows an example of a partitioned belief space.

Three different belief spaces are shown:

- S1 believes that Mary hit Bill.
- S2 believes that Sue hit Bill.
- S3 believes that someone hit Bill. It is important to be able to handle incomplete beliefs of this kind, since they frequently serve as the basis for questions, such as, in this case. "Who hit Bill?"

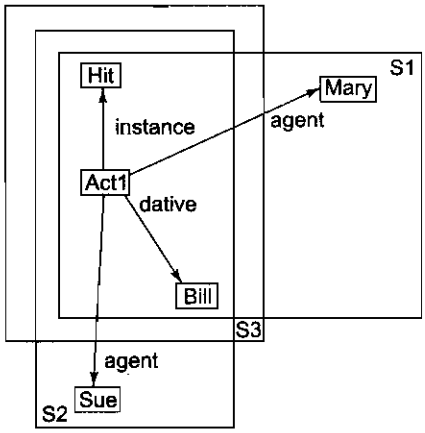


Fig. 15.20 *A Partitioned Semantic Net Showing Three Belief Spaces*

15.4.3 Using Goals and Plans for Understanding

Consider the text

> John was anxious to get his daughter's new bike put together before Christmas Eve. He looked high and low for a screwdriver.

To understand this story, we need to recognize that John had

1. A goal, getting the bike put together.
2. A plan, which involves putting together the various subparts until the bike is complete. At least one of the resulting subplans involves using a screwdriver to screw two parts together.

Some of the common goals that can be identified in stories of all softs (including children's stories, newspaper reports, and history books) are

- Satisfaction goals, such as sleep, food, and water.
- Enjoyment goals, such as entertainment and competition.
- Achievement goals, such as possession, power, and status.
- Preservation goals, such as health and possessions.
- Pleasing goals, which involve satisfying some other kind of goal for someone else.
- Instrumental goals, which enable preconditions for other, higher-level goals.

To achieve their goals, people exploit plans. In Chapter 13, we talked about several computational representations of plans. These representations can be used to support natural language processing, particularly if they are combined with a knowledge base of operators and stored plans that describe the ways that people often accomplish common goals. These stored operators and plans enable an understanding system to form a coherent representation of a text even when steps have been omitted, since they specify things that must have occurred in the complete story. For example, to understand this simple text about John, we need to make use of the fact that John was exploiting the operator USE (by A of P to perform G), which can be described as:

USE(A, P, G):
precondition: KNOW-WHAT(A, LOCATION(P))
NEAR(A, P)
HAS-CONTROL-OF(A, P)
READY(P)
postcondition: DONE(G)

In other words, for A to use P to perform G, A must know the location of P, A must be near P, A must have control of P (for example, I cannot use a screwdriver that you are holding and refuse to give to me), and P must be ready for use (for example, I cannot use a broken screwdriver).

In our story, John's plan for constructing the bike includes using a screwdriver. So he needs to establish the preconditions for that use. In particular, he needs to know the location of the screwdriver. To find that out, he makes use of the operator LOOK-FOR:

LOOK-FOR(A, P):
precondition: CAN-RECOGNIZE(A, P)
postcondition: KNOW-WHAT(A, LOCATION(P))

A story understanding program can connect the goal of putting together the bike with the activity of looking for a screwdriver by recognizing that John is looking for a screwdriver so that he can use it as part of putting the bike together.

Often there are alternative operators or plans for achieving the same goal. For example, to find out where the screwdriver was, John could have asked someone. Thus the problem of constructing a coherent interpretation of a text or a discourse may involve considering many partial plans and operators.

Plan recognition has served as the basis for many understanding programs. PAM [Wilensky, 1981] is an early example; it translated stories into a CD representation.

Another such program was BORIS [Dyer, 1983]. BORIS used a memory structure called the Thematic Abstraction Unit to organize knowledge about plans, goals, interpersonal relationships, and emotions. For other examples, see Allen and Perrault [1980] and Sidner[1985].

15.4.4 Speech Acts

Language is a form of behavior. We use it as one way to accomplish our goals. In essence, we make communicative plans in much the same sense that we make plans for anything else [Austin, 1962]. In fact, as we just saw in the example above, John could have achieved his goal of locating a screwdriver by asking someone where it was rather than by looking for it. The elements of communicative plans are called *speech acts* [Searle, 1969]. We can axiomatize speech acts just as we axiomatized other operators in the previous section, except that we need to make use of modal operators that describe states of belief, knowledge, wanting, etc. For example, we can define the basic speech act *A* INFORM *B* of *P* as follows:

INFORM(*A*, *B*, *P*)
precondition: BELIEVE(*A*, *P*)
KNOW-WHAT(*A*, LOCATION(*B*))
postcondition: BELIEVE(*B*, BELIEVE(*A*, *P*))
BELIEVE-IN(*B*, *A*) → BELIEVER, (*B*, *P*)

To execute this operation, *A* must believe *P* and *A* must know where *B* is. The result of this operator is that *B* believes that *A* believes *P*, and if *B* believes in the truth of what *A* says, then *B* also believes *P*.

We can define other speech acts similarly. For example, we can define ASK-WHAT (in which *A* asks *B* the value of some predicate *P*):

ASK-WHAT(*A*, *B*, *P*):
precondition: KNOW-WHAT(*A*, LOCATION(*B*))
KNOW-WHAT(*B*, *P*)
WILLING-TO-PERFORM
(*B*, INFORM(*B*, *A*, *P*))
postcondition: KNOW-WHAT(*A*, *P*)

This is the action that John could have performed as an alternative way of finding a screwdriver.

We can also define other speech acts, such as *A* REQUEST *B* to perform *R*:

REQUEST(*A*, *B*, *R*)
precondition: KNOW-WHAT(*A*, LOCATION(*B*))
CAN-PERFORM(*B*, *R*)
WILLING-TO-PERFORM(*B*, *R*)
postcondition: WILL(PERFORM(*B*, *R*))

15.4.5 Conversational Postulates

Unfortunately, this analysis of language is complicated by the fact that we do not always say exactly what we mean. Instead, we often use *indirect speech acts*, such as "Do you know what time it is?" or "It sure is cold in

here." Searle [1975] presents a linguistic theory of such indirect speech acts. Computational treatments of this phenomenon usually rely on models of the speaker's goals and of ways that those goals might reasonably be achieved by using language. See, for example, Cohen and Perrault [1979].

Fortunately, there is a certain amount of regularity in people's goals and in the way language can be used to achieve them. This regularity gives rise to a set *of conversational postulates,* which are rules about conversation that are shared by all speakers. Usually these rules are followed. Sometimes they are not, but when this happens, the violation of the rules communicates something in itself. Some of these conversational postulates are:

- *Sincerity Conditions*—For a request by *A* of *B* to do *R* to be sincere, *A* must want *B* to do *R*, *A* must assume *B* can do *R*, *A* must assume *B* is willing to do *R*, and *A* must believe that *B* would not have done *R* anyway. If *A* attempts to verify one of these conditions by asking a question of *B*, that question should normally be interpreted by *B* as equivalent to the request *R*. For example,

  A: Can you open the door?

- *Reasonableness Conditions*—For a request by *A* of *B* to do *R* to be reasonable, *A* must have a reason for wanting *R* done, *A* must have a reason for assuming that *B* can do *R*, *A* must have a reason for assuming that *B* is willing to do *R*, and *A* must have a reason for assuming that *B* was not already planning to do *R*. Reasonableness conditions often provide the basis for challenging a request. Together with the sincerity conditions described above, they account for the coherence of the following interchange:

  A: Can you open the door?
  B: Why do you want it open?

- *Appropriateness Conditions*—For a statement to be appropriate, it must provide the correct amount of information, it must accurately reflect the speaker's beliefs, it must be concise and unambiguous, and it must be polite. These conditions account for *A*'s response in the following interchange:

  A: Who won the race?
  B: Someone with long, dark hair.
  A: I thought you knew all the runners.

  A inferred from *B*'s incomplete response that *B* did not know who won the race, because if *B* had known she would have provided a name.
  Of course, sometimes people "cop out" of these conventions. In the following dialogue, *B* is explicitly copping out:

  A: Who is going to be nominated for the position?
  B: I'm sorry, I cannot answer that question.

But in the absence of such a cop out, and assuming a cooperative relationship between the parties to a dialogue, the shared assumption of these postulates greatly facilitates communication. For a more detailed discussion of conversational postulates, see Grice [1975] and Gordon and Lakoff [1975].

We can axiomatize these conversational postulates by augmenting the preconditions for the speech acts that we have already defined. For example, we can describe the sincerity conditions by adding the following clauses to the precondition for REQUEST(*A*, *B*, *R*):

WANT(*A*, PERFORM(*B*, *R*))
BELIEVE(*A*, CAN-PERFORM(*B*, *R*))
BELIEVE(*A*, WILLING-TO-PERFORM(*B*, *R*))
BELIEVE(*A*, ¬WILL(PERFORM(*B*, *R*)))

If we assume that each participant in a dialogue is following these conventions, then it is possible to infer facts about the participants' belief states from what they say. Those facts can then be used as a basis for constructing a coherent interpretation of a discourse as a whole.

To summarize, we have just described several techniques for representing knowledge about how people act and talk. This knowledge plays an important role in text and discourse understanding, since it enables an understander to fill in the gaps left by the original writer or speaker. It turns out that many of these same mechanisms, in particular those that allow us to represent explicitly the goals and beliefs of multiple agents, will also turn out to be useful in constructing distributed reasoning systems, in which several (at least partially independent) agents interact to achieve a single goal. We come back to this topic in Section 16.3.

15.5 STATISTICAL NATURAL LANGUAGE PROCESSING

Long sentences most often give rise to ambiguities when conventional grammars are used to process the same. The processing of such sentences may yield a large number of analyses. It is here that the statistical information extracted from a large corpus of the concerned language can aid in disambiguation. Since a complete study of how statistics can aid natural language processing cannot be discussed, we try to highlight some issues that will kindle the reader's interest in the same.

15.5.1 Corpora

The term "*corpus*" is derived from the Latin word meaning "*body*". The term could be used to define a collection of written text or spoken words of a language. In general a corpus could be defined as a large collection of segments of a language. These segments are selected and ordered based on some explicit linguistic criteria so that they may be used to depict a sample of that language. Corpora may be available in the form of a collection of raw text or in a more sophisticated annotated or marked-up form wherein information about the words is also included to ease the process of language processing.

Several kinds of corpora exist. These include ones containing written or spoken language, new or old texts, texts from either one or different languages. Textual content could mean the content of a complete book or books, newspapers, magazines, web pages, journals, speeches, etc. The British National Corpus (BNC), for instance is said to have a collection of around a hundred million written and spoken language samples. Some corpora may contain texts on a particular domain of study or a dialect. Such corpora are called *Sublanguage Corpora*. Others may focus specifically to select areas like medicine, law, literature, novels, etc.

Rather than just being a collection of raw text some corpora contain extra information regarding their content. The words are labeled with a linguistic tag that could mean the part of speech of the word or some other semantic category. Such corpora are said to be annotated. A *Treebank* is an annotated corpus that contains parse trees and other related syntactic information. The Penn Treebank made available by the University of Pennsylvania is a typical example of such a corpus. Naturally the creation of such annotation requires a lot of extra effort involving linguists.

Some corpora contain a collection of texts which have been translated into one or several other languages. These corpora are referred to as *parallel corpora* and find their use in language processing applications that involve translation capabilities. They facilitate the translation of words, phrases and sentences from one language to another. Tagging of corpora is done part manually and part automatically.

A *concordance* is a typical term used with reference to corpora. Concordance in general is an index or list of the important words in a text or a group of texts. Most often when we refer to a corpus, we are looking for concordances. Concordances can give us the notion of how often a word occurs (frequency), or, even, does not occur.

Another term that we often come across when we deal with corpus processing is a *collocation*. A collocation is a collection of words that are often observed together in a text. If we are talking about Christmas, then the words *Christmas gifts* forms a collocation. A *chain smoker*, *a hard nut*, *extremely beautiful*, are all examples

of collocations. Note that we do not generally refer to a smoker as an *intense* or *severe smoker*, nor do we remark someone to be *tremendously beautiful*. Collocations can thus aid us in the search for the apt words.

15.5.2 Counting the elements in a Corpus

Counting the number of words in a corpus as also the distinct words in it can yield valuable information regarding the probability of the occurrence of a word given an incomplete string in the language under consideration. These probabilities can be used to predict a word that will follow. How should counting be done depends on the application scenario. Should the punctuation marks like , (comma), ; (semicolon) and the period (.) be treated as a word or not has to be decided. The question mark (?) allows us to understand that something is being asked. Other issues in counting are whether to treat words like *In* and *in* (case sensitization), *book* and *books* (singular and plural) as distinct ones. Thus we arrive at two terms called *Types* and *Tokens*. The former means the number of distinct words in the corpus while the latter stands for the total number of words in the corpus. In the last sentence, (the one earlier to this), for example, we have 14 types and 24 tokens.

15.5.3 N-Grams

N-grams are basically sequences of *N* words or strings, where *N* could assume the value 1,2,3, and so on. When *N*=1, we call it a unigram (just one word). *N*=2 makes a bigram (a sequence of two words). Similarly we have trigrams, tetragrams and so on. Let's see in what way these *N*-grams make sense to us.

Different words in a corpus have their own frequency (i.e. the number of times they occur) in a given corpus. Some words have a high frequency like the article "*the*". As an example let us take the number of occurrences of words in the novel – *The Scarlet Pimpernel* by Baroness Orczy (It also provides for good reading! You can download the text from http://www.gutenberg.org). The novel has around 87163 words and 8822 types or word forms. Some typical words within are listed in the Table 15.1 along with their probabilities of occurrence in the text.

Table 15.1 *Frequencies and probabilities of some words in a corpus*

Word	*Word Frequency*	*Probability = (Word Frequency)/ Total number of words*
the	4508	0.051
of	2474	0.028
and	2353	0.027
to	2267	0.026
a	1559	0.018
her	1137	0.013
had	1077	0.012
she	935	0.0107
It	444	0.005
said	399	0.0046
man	174	0.002
Scarlet	98	0.0011
woman	66	0.00076
beautiful	39	0.00045
fool	18	0.00002
However	19	0.00002

Here we look at the probability of just one word in the corpus. Let us go a step further and ask - *What is the probability of a word being followed by another?*

(of *the* and *it*) to guess that the word *the* is a better candidate. Note that *the* has higher probability. But things do not always work this way. If we consider the segment of a sentence –

A very wealthy…

If we assume that the high frequency word *the* will follow the word *wealthy* it would lead to a syntactic error. The word *man* with a probability much less than *the* seems more appropriate. It thus seems that the next word is dependent on the previous one. Finding probabilities of all such words in the corpus that follow the word *wealthy* and then choosing the best of them may lead to the construction of a more appropriate sentence. Thus as we move through a sentence we could keep looking at a *two-word* window and predict the next or second word using probabilities of finding the second word, given the first word. This can be more formally written as–

$max(P(X|wealthy))$

or in plain English we find that word X which appears after *wealthy* and has the maximum probability of occurrence in this two-word sequence (viz. *wealthy* followed by *X*) among all other such words in the corpus.

If there are n words in a sentence and assuming the occurrence of each word at their appropriate places to be independent events, the probability $P(w_1,..., w_n)$ can be expressed using the chain rule as

$$P(W) = P(w_1) . P(w_2 | w_1) . P(w_3 | (w_1, w_2))... P(w_n | (w_1, w_2... w_{n-1}))$$

Computing this probability is far from simple. Observe that as we move to rightwards, the terms become more complex. The last term would naturally be the most complex to compute. A more practical approach to such chaining of probabilities could be to look at only one prior word at any given moment of processing. In other words, given a word we look for only the previous word to compute the probabilities. Since we look at only word pairs (viz. a single word previous to the one we are searching) this model is called the *bigram* model.

If we follow the bigram model of seeking the next word using the novel used as the corpus the word that would follow *Scarlet* would most aptly be *Pimpernel*. This is substantiated by the data on words that appear after the word *Scarlet* depicted in Table 15.2.

Table 15.2 *Frequencies of words following the word Scarlet*

Word following Scarlet	*Frequency*
Pimpernel	105
geranium	2
heels	1
waistcoat	1
flower	1
device	1
enigma	1

It can thus be assumed that when we refer to a previous word and find the probability of the next word, a more apt sentence is created. This is called a *Markov assumption*. Based on this, for a bigram model, the probability $P(w_n | (w_1.w_2... w_{n-1}))$ can be approximated to the product of all $P(w_i | w_{i-1})$ for i varying from *1* to n (n is the number of words in the sentence) i.e.

$$P(w_1^n) \approx \prod_{i=1}^{n} P(w_i|w_{i-1})$$

We could extend the concept from bigrams (taking into consideration only two words viz. the current and the previous) to trigrams (viz. taking the current word and the previous two words) and further on, to *tetragrams* (previous four words). The approximate probability of finding the next word in case of N-grams is given by

$$P(w_n|w_1^{n-1}) \approx P(w_n|w_{n-N+1}^{n-1})$$

w_1^{n-1} includes the words from w_1 to w_{n-1}. Similarly, w_{n-N+1}^{n-1} means words w_{n-N+1} to w_{n-1}.

It may now be interesting to note that the probability of the sentence –

He never told her and she had never cared to ask.

can be found using the bigram probability model as–

P(He never told her and she had never cared to ask.)

= $P(He|<nil>).P(never|He).P(told|never).P(her|told).P(and|her).P(she|and).P(had|she).P(never|had)\ P(cared\ |never).P(to|cared).P(ask|to)$.

= (207/67675).(3/512).(1/60).(10/31).(6/1137).(34/2353).(168/935).(6/1077).(1/60).(3/11).(3/2267)

Note that each probability term is calculated by finding the number of occurrences of the specific bigram and dividing it by the frequency of the previous word.

Thus,

P(she|and) = (Number of occurrence of the bigram ***and she***)/(Number of occurrences of the word ***and***)

Observe that the denominator could also be interpreted as the number of bigrams that start with the word ***and***.

When we wish to predict the next word given a word, we may find all the bigram frequencies starting with the given word and use the next word of that bigram that has highest frequency. The concept can be extended to higher grams viz. tri, tetra and finally *N*-grams.

So, what can we do with these *grams*? If we have a large corpus from which the related probabilities can be calculated, we could generate sentences and verify their correctness. Starting with one word we could predict what could be the next, and then do the same for the next word; always using the maximum probability to select the next word in the sequence.

Table 15.3 shows the bigram counts from our corpora for the sentence.

Table 15.3 *Bigram counts (The number in the bracket indicates the probability.)*

	He	*never*	*told*	*her*	*and*	*she*	*had*	*never*	*cared*	*to*	*ask*
He [207]	0(0)	3(0.014)	0(0)	0(0)	4(0.019)	0(0)	114(0.55)	3(0.0144)	0(0)	2(0.01)	0(0)
never [60]	0(0)	0(0)	1(0.02)	0(0)	0(0)	0(0)	3(0.05)	0(0)	1(0.02)	0(0)	0(0)
told [31]	0(0)	0(0)	0(0)	10(0.323)	0(0)	0(0)	0(0)	0(0)	0(0)	0(0)	0(0)
her [1137]	1(0.00008)	0(0)	0(0)	0(0)	6(0.005)	0(0)	1(0.00008)	0(0)	0(0)	19(0.0167)	0(0)
and [2353]	34(0.0144)	1(0.00004)	0(0)	26(0.011)	0(0)	34(0.014)	23(0.01)	1(0.00004)	0(0)	38(0.016)	0(0)
she [935]	0(0)	1(0.001)	1(0.001)	0(0)	3(0.003)	0(0)	168(0.18)	1(0.001)	2(0.002)	0(0)	0(0)
had [1077]	8(0.007)	6(0.006)	3(0.003)	0(0)	0(0)	9(0.008)	12(0.111)	6(0.006)	1(0.0001)	8(0.007)	0(0)
never [60]	0(0)	0(0)	1(0.02)	0(0)	0(0)	0(0)	3(0.05)	0(0)	1(0.02)	0(0)	0(0)
cared [11]	0(0)	0(0)	0(0)	0(0)	1(0.090)	0(0)	0(0)	0(0)	0(0)	3(0.273)	0(0)
to [2267]	0(0)	0(0)	0(0)	99(0.044)	1(0.00004)	0(0)	0(0)	0(0)	0(0)	0(0)	2(0.00008)
ask [12]	0(0)	0(0)	0(0)	0(0)	0(0)	0(0)	0(0)	0(0)	0(0)	0(0)	0(0)

Extending this concept we can look at trigrams where we use two words to predict the next one. In the example sentence above we could calculate the probability of a trigram as *P(and|told her)*.

15.5.4 Smoothing

While *N*-grams may be a fairly good way of predicting the next word, it does suffer from a major drawback – it banks heavily on the corpus which forms the basic training data. Any corpus is finite and there are bound to be many *N*-grams missing within it. There are numerous *N*-grams which should have had non-zero probability but are assigned a zero value instead. Observe such bigrams in Table 15.3 It is thus best if we could assign some non zero probability to circumvent the problem to some extent. This process is known as *smoothing*. Two known methods are described herein.

Add-One Smoothing

Let $P(w_i)$ be the normal unigram (single word) probability without smoothing (unsmoothed) and N be the total number of words in the corpus then,

$$P(w_i) = c(w_i)/\Sigma c(w_i) = c(w_i)/N$$

This is the simplest way of assigning non-zero probabilities. Before calculating the probabilities, the count *c* of each distinct word within the corpus is incremented by 1. Note the total of the counts of all words has now increased by D the number of distinct types of words in the language (vocabulary).

The new probability of a word after *add one smoothing* can now be computed as–

$$P_{add1}(w_i) = \{c(w_i)+1\}/(N+D)$$

The reader is urged to re-compute the probability using the information in Table 15.3 and inspect the fresh values.

Witten-Bell Discounting

The probability of unseen *N*-grams could be looked upon as things we saw once (for the first time). As we go through the corpus we do encounter new *N*-grams and finally are in a position to ascertain the number of unique *N*-grams. The event of encountering a new *N*-gram could be looked upon as a case of an (so far) unseen *N*-gram. This calls for computing the probability of an *N*-gram which has just been sighted. One may observe that the number of unique *N*-grams seen in the data is the same as the count, *H*, of the *N*-grams observed (so far) for the first time. Thus *(N+H)* would mean the sum of the words or tokens seen so far and the unique *N*-grams types in the corpus.

The total probability mass of all such N-grams (occurring for the first time i.e. having zero probability) could be estimated by computing *H/(N+H)*. This value stands for the probability of a new type of *N*-gram being detected. *H/(N+H)* is also the probability of unseen *N*-grams taken together. If *I* is the total number of *N*-grams that have never occurred so far (zero count), dividing the probability of unseen *N*-grams by *I* would distribute it equally among them. Thus the probability of an unseen *N*-gram could be written as–

$$P^u = H/I(N+H)$$

Since the total probability has to be 1, this extra probability distributed amongst unseen *N*-grams has to be scooped or discounted from other regions in the probability distribution. The probability of the seen *N*-grams is therefore discounted to aid the generation of the extra probability requirement for the unseen ones as:

$P_k^s = c_k/(N+H)$ where c_k is the (non-zero positive) count of k^{th} *N*-gram.

15.6 SPELL CHECKING

A Spell Checker is one of the basic tools required for language processing. It is used in a wide variety of computing environments including word processing, character or text recognition systems, speech recognition

and generation. Spell checking is one of the pre-processing formalities for most natural language processors. Studies on computer aided spell checking date back to the early 1960's and with the advent of it being applied to new languages, continue to be one of the challenging areas in information processing. Spell checking involves identifying words and non words and also suggesting the possible alternatives for its correction. Most available spell checkers focus on processing isolated words and do not take into account the context. For instance if you try typing –

"*Henry sar on the box*"

in Microsoft Word 2003 and find what suggestions it serves, you will find that the correct word *sat* is missing! Now try typing this –

"*Henry at on the box*"

Here you will find that the error remains undetected as the word *at* is spelt correctly as an isolated word. Observe that context plays a vital role in spell checking.

15.6.1 Spelling Errors

Damerau (1964) conducted a survey on misspelled words and found that most of the non words were a result of single error misspellings. Based on this survey it was found that the three causes of error are:

- *Insertion*: Insertion of an extra letter while typing. E.g. *maximum* typed as *maxiimum*. The extra *i* has been inserted within the word.
- *Deletion*: A case of a letter missing or not typed in a word. E.g. *netwrk* instead of *network*.
- *Substitution*: Typing of a letter in place of the correct one as in *intellugence* wherein the letter *i* has been wrongly substituted by *u*.

Spelling errors may be classified into the following types –

Typographic errors:

As the name suggests, these errors are those that are caused due to mistakes committed while typing. A typical example is *netwrk* instead of *network*.

Orthographic errors:

These, on the other hand, result due to a lack of comprehension of the concerned language on part of the user. Example of such spelling errors are *arithmetic*, *wellcome* and *accomodation*.

Phonetic errors:

These result due to poor cognition on part of the listner. The word *rough* could be spelt as *ruff* and *listen* as *lisen*. Note that both the misspelled words *ruff* and *lisen* have the same phonetic pronunciation as their actual spellings. Such errors may distort misspelled words more than typographic editing actions that cause a misspelling (viz. insertion, deletion, transposition, or substitution error) as in case of *ruff*. Words like *piece*, *peace* and *peas, reed and read and quite and quiet* may all be spelt correctly but can lead to confusion depending on the context.

15.6.2 Spell Checking Techniques

One could imagine a naïve spell checker as a large corpus of correct words. Thus if a word in the text being corrected does not match with one in the corpus then it results in a spelling error. An exhaustive corpus would of course be a mandatory requirement.

Spell checking techniques can be broadly classified into three categories –

(a) [illegible]

This process involves the detection of misspelled words or non-words. For example –

The word *soper* is a non-word; its correct form being *super* (or maybe *sober*).

The most commonly used techniques to detect such errors are the N-gram analysis and Dictionary look-up. As discussed earlier, N-gram techniques make use of the probabilities of occurrence of N-grams in a large corpus of text to decide on the error in the word. Those strings that contain highly infrequent sequences are treated as cases of spelling errors. Note that in the context of spell checkers we take N-grams to be a sequence of letters (alphabet) rather than words. Here we try to predict the next letter (alphabet) rather than the next word. These techniques have often been used in text (handwritten or printed) recognition systems which are processed by an Optical Character Recognition (OCR) system. The OCR uses features of each character such as the curves and the loops made by them to identify the character. Quite often these OCR methods lead to errors. The number 0, the alphabet O and D are quite often sources of errors as they look alike. This calls for a spell checker that can post-process the OCR output. One common N-gram approach uses tables to predict whether a sequence of characters does exist within a corpora and then flags an error. Dictionary look-up involves the use of an efficient dictionary lookup coupled with pattern-matching algorithms (such as hashing techniques, finite state automata, etc.), dictionary partitioning schemes and morphological processing methods.

(b) Isolated-Word Error Correction:

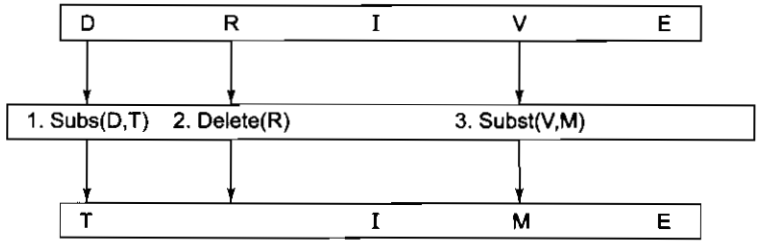
This process focuses on the correction of an isolated non-word by finding its nearest and meaningful word and makes an attempt to rectify the error. It thus transforms the word *soper* into *super* by some means but without looking into the context.

This correction is usually performed as a context independent suggestion generation exercise. The techniques employed herein include the minimum edit distance techniques, similarity key techniques, rule-based methods, N-gram, probabilistic and neural network based techniques (Kukich 1992).

Isolated-word error correction may be looked upon as a combination of three sub-problems – Error detection, Candidate (Correct word) generation and Ranking of the correct candidates. Error detection as already mentioned could use either of the dictionary or the N-gram approaches. The possible correct candidates are found using a dictionary or by looking-up a pre-processed database of correct N-grams. Ranking of these candidates is done by measuring the lexical or similarity distance between the misspelled word and the candidate.

Minimum Edit Distance Technique

Wagner [1974] defined the minimum edit distance between the misspelled word and the possible correct candidate as the minimum number of edit operations needed to transform the misspelled word to the correct candidate. By edit operations we mean – insertions, deletions and substitutions of a single character (alphabet) to transform one word to the other. The minimum number of such operations required to effect the transform is commonly known as the *Levenshtein* distance named after Vladimir Levenshtein who first used this metric as a distance. As an example inspect the way in which you could transform the word *drive* (below) to the word *time* and arrive at the distance 3 between them.



A variant of the Levenshtein distance is the Damerau–Levenshtein distance which also takes into account the transposition of two characters in addition to insertion, deletion and substitution.

(c) Context dependent Error detection and correction:

These processes try, in addition to detect errors, try to find whether the corrected word fits into the context of the sentence. These are naturally more complex to implement and require more resources than the previous method. How would you correct the wise words of Lord Buddha –

"Peace comes from within"

if it were typed as –

"Piece comes from within"?

Note that the first word in both these statements is a correct word.

This involves correction of real-word errors or those that result in another valid word. Non-word errors that have more than one potential correction also fall in this category. The strategies commonly used find their basis on traditional and statistical natural language processing techniques.

15.6.3 Soundex Algorithm

The Soundex algorithm can be effectively used as a simple phonetic based spell checker. It makes use of rules found using the phonetics of the language in question. We discuss this algorithm with reference to English.

Developed by Robert Russell and Margaret Odell in the early 20th century, the Soundex algorithm uses a code to check for the closest word. The code was used to index names in the U.S. census. The code for a word consists of its first letter followed by three numbers that encode the remaining consonants. Those consonants that generate the same sound have the same number. Thus the labials *B*, *F*, *P* and V imply the same number viz. 1.

Here is the algorithm –

- Remove all punctuation marks and capitalize the letters in the word.
- Retain the first letter of the word.
- Remove any occurrence of the letters – *A, E, I, O, U, H, W, Y* apart from the very first letter.
- Replace the letters (other than the first) by the numbers shown in Table 15.4.
- If two or more adjacent letters, not separated by vowels, have the same numeric value, retain only one of them.
- Return the first four characters; pad with zeroes if there are less than four.

Table 15.4 *Substitutions for generating the Soundex code*

Letter(s)	*Substitute with Integer*
B,F,P,V	1
C,G,J,K,S,X,Z	2
D,T	3
L	4
M,N	5
R	6

Nominally the Soundex code contains –

First letter of word, Code 1, Code 2, Code 3. Table 15.5 shows the Soundex code for a few words.

Table 15.5 *Soundex codes for some words*

Word	*Soundex Code*
Grate, great	G630
Network, network	N362
Henry, Henary	H560
Torn	T650
Worn	W650
Horn	H650

Note that the last three words are different only in the starting alphabet. This algorithm can thus be used to measure the similarity of two words. This measure can then be used to find possible good candidates for correction to be effected for a misspelled word such as *rorn* (viz. torn, worn, horn).

SUMMARY

In this chapter, we presented a brief introduction to the surprisingly hard problem of language understanding. Recall that in Chapter f4, we showed that at least one understanding problem, line labeling, could effectively be viewed as a constraint satisfaction problem. One interesting way to summarize the natural language understanding problem that we have described in this chapter is to view it too as a constraint satisfaction problem. Unfortunately, many more kinds of constraints must be considered, and even when they are all exploited, it is usually not possible to avoid the guess and search part of the constraint satisfaction procedure. But constraint satisfaction does provide a reasonable framework in which to view the whole collection of steps that together create a meaning for a sentence. Essentially each of the steps described in this chapter exploits a particular kind of knowledge that contributes a specific set of constraints that must be satisfied by any correct final interpretation of a sentence.

Syntactic processing contributes a set of constraints derived from the grammar of the language. It imposes constraints such as:

- Word order, which rules out, for example, the constituent, "manager the key," in the sentence, "I gave the apartment manager the key."
- Number agreement, which keeps "trial run" from being interpreted as a sentence in "The first trial run was a failure."
- Case agreement, which rules, out, for example, the constituent, "me and Susan gave one to Bob," in the sentence, "Mike gave the program to Alan and me and Susan gave one to Bob."

Semantic processing contributes an additional set of constraints derived from the knowledge it has about entities that can exist in the world. It imposes constraints such as:

- Specific kinds of actions involve specific classes of participants. We thus rule out the baseball field meaning of the word "diamond" in the sentence, "John saw Susan's diamond shimmering from across the room."
- Objects have properties that can take on values from a limited set. We thus rule out Bill's mixer as a component of the cake in the sentence, "John made a huge wedding cake with Bill's mixer."

Discourse processing contributes a further set of constraints that arise from the structure of coherent discourses. These include:

- The entities involved in the sentence must either have been introduced explicitly or they must be related to entities that were. Thus the word "it" in the discourse "John had a cold. Bill caught it," must refer to John's cold. This constraint can propagate through other constraints. For example, in this case, it can be used to determine the meaning of the word "caught" in this discourse, in contrast to its meaning in the discourse, "John threw the ball. Bill caught it."
- The overall discourse must be coherent. Thus, in the discourse, "I needed to deposit some money, so I went down to the bank," we would choose the financial institution reading of bank over the river bank reading. This requirement can even cause a later sentence to impose a constraint on the interpretation of an earlier one, as in the discourse, "I went down to the bank. The river had just flooded, and I wanted to see how bad things were."

And finally, pragmatic processing contributes yet another set of constraints. For example,

- The meaning of the sentence must be consistent with the known goals of the speaker. So, for example, in the sentence, "Mary was anxious to get the bill passed this session, so she moved to table it," we are forced to choose the (normally British) meaning of table (to put it on the table for discussion) over the (normally American) meaning (to set it aside for later).

There are many important issues in natural language processing that we have barely touched on here. To learn more about the overall problem, see Allen [1987], Cullingford [1986], Dowty *et al.* [1985],and Grosz *et al.* [1986]. For more information on syntactic processing, see Winograd [1983] and King [1983]. See Joshi *et al.* [1981] for more discussion of the issues involved in discourse understanding. Also, we have restricted our discussion to natural language understanding. It is often useful to be able to go the other way as well, that is, to begin with a logical description and render it into English. For discussions of natural language generation systems, see McKeown and Swartout [1987] and McDonald and Bole [1988]. By combining understanding and generation systems, it is possible to attack the problem of *machine translation, by* which we understand text written in one language and then generate it in another language. See Slocum [1988], Nirenburg [1987], Lehrberger and Bourbeau [1988], and Nagao [1989] for discussions of a variety of approaches to this problem.

We have also seen how statistical methods come to the aid of natural language processing. The use of a large corpus and the frequency and sequence of occurrence of words can be used to decide and predict the correctness of a given text. This can also be used for language generation to a certain extent.

Natural language processing also entails spell checking as a preprocessing exercise. This chapter introduced some common spelling errors, checking and suggestion generation methods. A good discussion on spell checkers can be found in Kuckich's paper entitled "Techniques for Automatically Correcting Words in Text", ACM Computing Surveys, Vol. 24, No. 4, December 1992, pp. 377-439. Other references include F. J. Damerau, "A technique for computer detection and correction of spelling errors", Communications of the ACM Vol.7, No. 3(Mar.), 1964, pp.171-176, Gonzalo Navarro, "A guided tour to Approximate String Matching", ACM Computing Surveys Vol.33, No.1 (Mar.), 2001, pp. 31-88, J. J. Pollock, and A. Zamora, "Automatic spelling correction in scientific and scholarly text", Communications of the ACM Vol. 27, No. 4 (Apr.), 1984, pp.358-368, J.R. Ullmann, "A binary n-gram technique for automatic correction of substitution, deletion, insertion, and reversal errors in words", Computer Journal, Vol. 20, No.2, 1977, pp.141-147, D. Jurafsky, and J.H. Martin, An Introduction to Natural Language Processing, Computational Linguistics and Speech Recognition, Prentice Hall Inc, New Jersey, U.S.A., 2000. As a tail ender one must bear in mind that once a spell checker has been designed for a language, the same checker need not always work well for another. Considerable effort may have to be put in before the realization of the same for a new language. The paper by M. Das, S.Borgohain, J.Gogoi, S.B.Nair, "Design and Implementation of a Spell Checker for Assamese", Proceedings of the Language Engineering Conference, 2002, IEEE CS Press, pp.156-162, throws more insights into this aspect.

Inquisitive readers could, after reading Chapter 24, go through the paper entitled "An Artificial Immune System Based Approach for English Grammar Checking" authored by Akshat Kumar and S. B. Nair, appearing in the book Artificial Immune Systems: Proceedings of the 6th International Conference on AIS, ICARIS 2007, Santos, Brazil, 2007, Eds. Leandro N. de Castro, Fernando J. Von Zuben, Helder Knidel, Springer, 2007, pp. 348-357, and ruminate on off-the-track approaches to natural language processing

EXERCISES

1. Consider the sentence

   The old man's glasses were filled with sherry.

   What information is necessary to choose the correct meaning for the word "glasses"? What information suggests the incorrect meaning?
2. For each of the following sentences, show a parse tree. For each of them, explain what knowledge, in addition to the grammar of English, is necessary to produce the correct parse. Expand the grammar of Fig. 15.6 as necessary to do this.
   - John wanted to go to the movie with Sally.
   - John wanted to go to the movie with Robert Redford.
   - I heard the story listening to the radio.
   - I heard the kids listening to the radio.
   - All books and magazines that deal with controversial topics have been removed from the shelves.
   - All books and magazines that come out quarterly have been removed from the shelves.
3. In the following paragraph, show the antecedents for each of the pronouns. What knowledge is necessary to determine each?

   John went to the store to buy a shirt. The salesclerk asked him if he could help him. He said he wanted a blue shirt. The salesclerk found one and he tried it on. He paid for it and left.
4. Consider the following sentence:
   Put the red block on the blue block on the table.
   (a) Show all the syntactically valid parses of this sentence. Assume any standard grammatical formalism you like.
   (b) How could semantic information and world knowledge be used to select the appropriate meaning of this command in a particular situation?

   After you have done this, you might want to look at the discussion of this problem in Church and Patil [1982].
5. Each of the following sentences is ambiguous in at least two ways. Because of the type of knowledge represented by each sentence, different target languages may be useful to characterize the different meanings. For each of the sentences, choose an appropriate target language and show how the different meanings would be represented:
   - Everyone doesn't know everything.
   - John saw Mary and the boy with a telescope.
   - John flew to New York.
6. Write an ATN grammar that recognizes verb phrases involving auxiliary verbs. The grammar should handle such phrases as
   - "went"
   - "should have gone"
   - "had been going"

- "would have been going"
- "would go"

Do not expect to produce an ATN that can handle all possible verb phrases. But do design one with a reasonable structure that handles most common ones, including the ones above. The grammar should create structures that reflect the structures of the input verb phrases.

7. Show how the ATN of Figs 15.8 and 15.9 could be modified to handle passive sentences. :
8. Write the rule "S → NP VP" in the graph notation that we defined in Section 15.2.3. Show how unification can be used to enforce number agreement between the subject and the verb.
9. Consider the problem of providing an English interface to a database of employee records.
   (a) Write a semantic grammar to define a language for this task.
   (b) Show a parse, using your grammar, of each of the two sentences
   - What is Smith's salary?
   Tell me who Smith's manager is.
   (c) Show parses of the two sentences of part (b) using a standard syntactic grammar of English. Show the fragment of the grammar that you use.
   (d) How do the parses of parts (b) and (c) differ? What do these differences say about the differences between syntactic and semantic grammars?
10. How would the following sentences be represented in a case structure:
    (a) The plane flew above the clouds
    (b) John flew to New York
    (c) The co-pilot flew the plane
11. Both case grammar and conceptual dependency produce representations of sentences in which noun phrases are described in terms of their semantic relationships to the verb. In what ways are the two approaches similar? In what ways are they different? Is one a more general version of the other? As an example, compare the representation of the sentence

    John broke the window with a hammer

    in the two formalisms
12. Use compositional semantics and a knowledge base to construct a semantic interpretation of each of the following sentences:
    (a) A student deleted my file
    (b) John asked Mary to print the file

    To do this, you will need to do all the following things:
    - Define the necessary knowledge base objects
    - Decide what the output of your parser will be assumed to be
    - Write the necessary semantic interpretation rules
    - Show how the process proceeds
13. Show how conversational postulates can be used to get to the most common, coherent interpretation of each of the following discourses:
    (a) A: Do you have a comb?
    (b) A: Would Jones make a good programmer? B: He's a great guy. Everyone likes him
    (c) A (in a store): Do you have any money? B (A's friend): What do you want to buy?
14. Winograd and Flores [1986] present an argument that it is wrong to attempt to make computers understand language. Analyze their arguments in light of what was said in this chapter.
15. Gather text from known and reliable sources and make your own corpus. Analyze the corpus by finding the number and the different types of words within and their unigram and bigram probabilities.
16. Using the information from exercise 15, try generating correct sentences using N-grams.
17. Explain how you would use the above corpus as a database for spell checking?

CHAPTER 17

LEARNING

That men do not learn very much from the lessons of history is the most important of all the lessons of history.

—**Aldous Huxley**
(1894–1963), American Writer and Author

17.1 WHAT IS LEARNING?

One of the most often heard criticisms of AI is that machines cannot be called intelligent until they are able to learn to do new things and to adapt to new situations, rather than simply doing as they are told to do. There can be little question that the ability to adapt to new surroundings and to solve new problems is an important characteristic of intelligent entities. Can we expect to see such abilities in programs? Ada Augusta, one of the earliest philosophers of computing, wrote that

> The Analytical Engine has no pretensions whatever to *originate* anything. It can do whatever we *know how to order it* to perform. [Lovelace, 1961]

This remark has been interpreted by several AI critics as saying that computers cannot learn. In fact, it does not say that at all. Nothing prevents us from telling a computer how to interpret its inputs in such a way that its performance gradually improves.

Rather than asking in advance whether it is possible for computers to "learn," it is much more enlightening to try to describe exactly what activities we mean when we say "learning" and what mechanisms could be used to enable us to perform those activities. Simon [1983] has proposed that learning denotes

> ...changes in the system that are adaptive in the sense that they enable the system to do the same task or tasks drawn from the same population more efficiently and more effectively the next time.

As thus defined, learning covers a wide range of phenomena. At one end of the spectrum is *skill refinement*. People get better at many tasks simply by practicing. The more you ride a bicycle or play tennis, the better you get. At the other end of the spectrum lies *knowledge acquisition*. As we have seen, many AI programs draw

heavily on knowledge as their source of power. Knowledge is generally acquired through experience, and such acquisition is the focus of this chapter.

Knowledge acquisition itself includes many different activities. Simple storing of computed information, or *rote learning*, is the most basic learning activity. Many computer programs, e.g., database systems, can be said to "learn" in this sense, although most people would not call such simple storage learning. However, many AI programs are able to improve their performance substantially through rote-learning techniques, and we will look at one example in depth, the checker-playing program of Samuel [1963].

Another way we learn is through taking advice from others. Advice taking is similar to rote learning, but high-level advice may not be in a form simple enough for a program to use directly in problem-solving. The advice may need to be first *operationalized*, a process explored in Section 17.3.

People also learn through their own problem-solving experience. After solving a complex problem, we remember the structure of the problem and the methods we used to solve it. The next time we see the problem, we can solve it more efficiently. Moreover, we can generalize from our experience to solve related problems more easily. In contrast to advice taking, learning from problem-solving experience does not usually involve gathering new knowledge that was previously unavailable to the learning program. That is, the program remembers its experiences and generalizes from them, but does not add to the transitive closure[1] of its knowledge, in the sense that an advice-taking program would, i.e., by receiving stimuli from the outside world. In large problem spaces, however, efficiency gains are critical. Practically speaking, learning can mean the difference between solving a problem rapidly and not solving it at all. In addition, programs that learn through problem-solving experience may be able to come up with qualitatively better solutions in the future.

Another form of learning that does involve stimuli from the outside is *learning from examples*. We often learn to classify things in the world without being given explicit rules. For example, adults can differentiate between cats and dogs, but small children often cannot. Somewhere along the line, we induce a method for telling cats from dogs based on seeing numerous examples of each. Learning from examples usually involves a teacher who helps us classify things by correcting us when we are wrong. Sometimes, however, a program can discover things without the aid of a teacher.

AI researchers have proposed many mechanisms for doing the kinds of learning described above. In this chapter, we discuss several of them. But keep in mind throughout this discussion that learning is itself a problem-solving process. In fact, it is very difficult to formulate a precise definition of learning that distinguishes it from other problem-solving tasks. Thus it should come as no surprise that, throughout this chapter, we will make extensive use of both the problem-solving mechanisms and the knowledge representation techniques that were presented in Parts I and II.

17.2 ROTE LEARNING

When a computer stores a piece of data, it is performing a rudimentary form of learning. After all, this act of storage presumably allows the program to perform better in the future (otherwise, why bother?). In the case of data caching, we store computed values so that we do not have to recompute them later. When computation is more expensive than recall, this strategy can save a significant amount of time. Caching has been used in AI programs to produce some surprising performance improvements. Such caching is known as *rote learning*.

In Chapter 12, we mentioned one of the earliest game-playing programs, Samuel's checkers program [Samuel, 1963]. This program learned to play checkers well enough to beat its creator. It exploited two kinds of learning: rote learning, which we look at now, and parameter (or coefficient) adjustment, which is described in Section 17.4.1. Samuel's program used the minimax search procedure to explore checkers game trees. As

[1] The transitive closure of a program's knowledge is that knowledge plus whatever the program can logically deduce from it.

is the case with all such programs, time constraints permitted it to search only a few levels in the tree. (The exact number varied depending on the situation.) When it could search no deeper, it applied its static evaluation function to the hoard position and used that score to continue its search of the game tree. When it finished searching the tree and propagating the values backward, it had a score for the position represented by the root of the tree. It could then choose the best move and make it. But it also recorded the board position at the root of the tree and the backed up score that had just been computed for it. This situation is shown in Fig. 17.1 (a).

Now suppose that in a later game, the situation shown in Fig. 17.1 (b) were to arise. Instead of using the static evaluation function to compute a score for position A, the stored value for A can be used. This creates the effect of having searched an additional several ply since the stored value for A was computed by backing up values from exactly such a search.

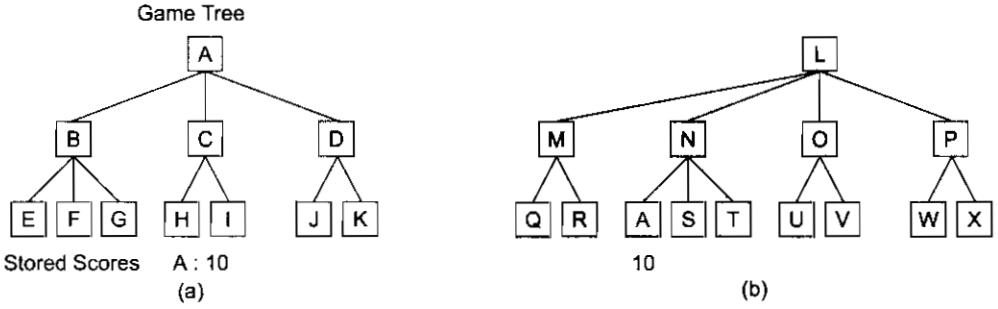


Fig.17.1 *Storing Backed-Up Values*

Rote learning of this sort is very simple. It does not appear to involve any sophisticated problem-solving capabilities. But even it shows the need for some capabilities that will become increasingly important in more complex learning systems. These capabilities include:

- *Organized Storage of Information*—In order for it to be faster to use a stored value than it would be to recompute it, there must be a way to access the appropriate stored value quickly. In Samuel's program, this was done by indexing board positions by a few important characteristics, such as the number of pieces. But as the complexity of the stored information increases, more sophisticated techniques are necessary.
- *Generalization*—The number of distinct objects that might potentially be stored can be very large. To keep the number of stored objects down to a manageable level, some kind of generalization is necessary. In Samuel's program, for example, the number of distinct objects that could be stored was equal to the number of different board positions that can arise in a game. Only a few simple forms of generalization were used in Samuel's program to cut down that number. All positions are stored as though White is to move. This cuts the number of stored positions in half. When possible, rotations along the diagonal are also combined. Again, though, as the complexity of the learning process increases, so too does the need for generalization.

At this point, we have begun to see one way in which learning is similar to other kinds of problem solving. Its success depends on a good organizational structure for its knowledge base.

17.3 LEARNING BY TAKING ADVICE

A computer can do very little without a program for it to run. When a programmer writes a series of instructions into a computer, a rudimentary kind of learning is taking place: The programmer is a sort of teacher, and the computer is a sort of student. After being programmed, the computer is now able to do something it previously could not. Executing the program may not be such a simple matter, however. Suppose the program is written

in a high-level language like LISP. Some interpreter or compiler must intervene to change the teacher's instructions into code that the machine can execute directly.

People process advice in an analogous way. In chess, the advice "fight for control of the center of the board" is useless unless the player can translate the advice into concrete moves and plans. A computer program might make use of the advice by adjusting its static evaluation function to include a factor based on the number of center squares attacked by its own pieces.

Mostow [1983] describes a program called FOO, which accepts advice for playing hearts, a card game. A human user first translates the advice from English into a representation that FOO can understand. For example, "Avoid taking points" becomes:

```
(avoid (take-points me) (trick))
```

FOO must *operationalize* this advice by turning it into an expression that contains concepts and actions FOO can use when playing the game of hearts. One strategy FOO can follow is to UNFOLD an expression by replacing some term by its definition. By UNFOLDing the definition of avoid, FOO comes up with:

```
(achieve (not (during (trick) (take-points me))))
```

FOO considers the advice to apply to the player called "me." Next, FOO UNFOLDs the definition of trick:

```
(achieve (not (during
                (scenario
                  (each p1 (players) (play-card p1))
                  (take-trick (trick-winner)))
                (take-points me))))
```

In other words, the player should avoid taking points during the scenario consisting of (1) players playing cards and (2) one player taking the trick. FOO then uses *case analysis* to determine which steps could cause one to take points. It rules out step 1 on the basis that it knows of no intersection of the concepts take-points and play-card. But step 2 could affect taking points, so FOO UNFOLDs the definition of take-points:

```
(achieve (not (there-exists c1 (cards-played)
                (there-exists c2 (point-cards)
                  (during (take (trick-winner) c1)
                    (take me c2))))))
```

This advice says that the player should avoid taking point-cards during the process of the trick-winner taking the trick. The question for FOO now is: Under what conditions does (take me c2) occur during (take (trick-winner) c1)? By using a technique called *partial match*, FOO hypothesizes that points will be taken if me = trick-winner and c2 = c1. It transforms the advice into:

```
(achieve (not (and (have-points (cards-played))
                   (= (trick-winner) me))))
```

This means "Do not win a trick that has points." We have not traveled very far conceptually from "avoid taking points," but it is important to note that the current vocabulary is one that FOO can understand in terms of actually playing the game of hearts. Through a number of other transformations, FOO eventually settles on:

```
(achieve (>= (and (in-suit-led (card-of me))
                  (possible (trick-has-points)))
             (low (card-of me)))
```

In other words, when playing a card that is the same suit as the card that was played first, if the trick possibly contains points, then play a low card. At last, FOO has translated the rather vague advice "avoid taking points" into a specific, usable heuristic. FOO is able to play a better game of hearts after receiving this advice. A human can watch FOO play, detect new mistakes, and correct them through yet more advice, such as "play high cards when it is safe to do so." The ability to operationalize knowledge is critical for systems that learn from a teacher's advice. It is also an important component of explanation-based learning, another form of learning discussed in Section 17.6.

17.4 LEARNING IN PROBLEM-SOLVING

In the last section, we saw how a problem-solver could improve its performance by taking advice from a teacher. Can a program get better *without* the aid of a teacher? It can, by generalizing from its own experiences.

17.4.1 Learning by Parameter Adjustment

Many programs rely on an evaluation procedure that combines information from several sources into a single summary statistic. Game-playing programs do this in their static evaluation functions, in which a variety of factors, such as piece advantage and mobility, are combined into a single score reflecting the desirability of a particular board position. Pattern classification programs often combine several features to determine the correct category into which a given stimulus should be placed. In designing such programs, it is often difficult to know *a priori* how much weight should be attached to each feature being used. One way of finding the correct weights is to begin with some estimate of the correct settings and then to let the program modify the settings on the basis of its experience. Features that appear to be good predictors of overall success will have their weights increased, while those that do not will have their weights decreased, perhaps even to the point of being dropped entirely.

Samuel's checkers program [Samuel, 1963] exploited this kind of learning in addition to the rote learning described above, and it provides a good example of its use. As its static evaluation function, the program used a polynomial of the form

$$c_1t_1 + c_2t_2 + \ldots + c_{16}t_{16}$$

The t terms are the values of the sixteen features that contribute to the evaluation. The c terms are the coefficients (weights) that are attached to each of these values. As learning progresses, the c values will change.

The most important question in the design of a learning program based on parameter adjustment is "When should the value of a coefficient be increased and when should it be decreased?" The second question to be answered is then "By how much should the value be changed?" The simple answer to the first question is that the coefficients of terms that predicted the final outcome accurately should be increased, while the coefficients of poor predictors should be decreased. In some domains, this is easy to do. If a pattern classification program uses its evaluation function to classify an input and it gets the right answer, then all the terms that predicted that answer should have their weights increased. But in game-playing programs, the problem is more difficult. The program does not get any concrete feedback from individual moves. It does not find out for sure until the end of the game whether it has won. But many moves have contributed to that final outcome. Even if the program wins, it may have made some bad moves along the way. The problem of appropriately assigning responsibility to each of the steps that led to a single outcome is known as the *credit assignment problem.*

Samuel's program exploits one technique, albeit imperfect, for solving this problem. Assume that the initial values chosen for the coefficients are good enough that the total evaluation function produces values

that are fairly reasonable measures of the correct score even if they are not as accurate as we hope to get them. Then this evaluation function can be used to provide feedback to itself. Move sequences that lead to positions with higher values can be considered good (and the terms in the evaluation function that suggested them can be reinforced).

Because of the limitations of this approach, however, Samuel's program did two other things, one of which provided an additional test that progress was being made and the other of which generated additional nudges to keep the process out of a rut:

- When the program was in learning mode, it played against another copy of itself. Only one of the copies altered its scoring function during the game; the other remained fixed. At the end of the game, if the copy with the modified function won, then the modified function was accepted. Otherwise, the old one was retained. If, however, this happened very many times, then some drastic change was made to the function in an attempt to get the process going in a more profitable direction.
- Periodically, one term in the scoring function' was eliminated and replaced by another. This was possible because, although the program used only sixteen features at any one time, it actually knew about thirty-eight. This replacement differed from the rest of the learning procedure since it created a sudden change in the scoring function rather than a gradual shift in its weights.

This process of learning by successive modifications to the weights of terms in a scoring function has many limitations, mostly arising out of its lack of exploitation of any knowledge about the structure of the problem with which it is dealing and the logical relationships among the problem's components. In addition, because the learning procedure is a variety of hill climbing, it suffers from the same difficulties as do other hill-climbing programs. Parameter adjustment is certainly not a solution to the overall learning problem. But it is often a useful technique, either in situations where very little additional knowledge is available or in programs in which it is combined with more knowledge-intensive methods. We have more to say about this type of learning in Chapter 18.

17.4.2 Learning with Macro-Operators

We saw in Section 17.2 how rote learning was used in the context of a checker-playing program. Similar techniques can be used in more general problem-solving programs. The idea is the same: to avoid expensive recomputation. For example, suppose you are faced with the problem of getting to the downtown post office. Your solution may involve getting in your car, starting it, and driving along a certain route. Substantial planning may go into choosing the appropriate route, but you need not plan about how to go about starting your car. You are free to treat START-CAR as an atomic action, even though it really consists of several actions: sitting down, adjusting the mirror, inserting the key, and turning the key. Sequences of actions that can be treated as a whole are called *macro-operators.*

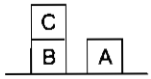
Macro-operators were used in the early problem-solving system STRIPS [Fikes and Nilsson, 1971; Fikes *et al.,* 1972]. We discussed the operator and goal structures of STRIPS in Section 13.2, but STRIPS also has a learning component. After each problem-solving episode, the learning component takes the computed plan and stores it away as a macro-operator, or MACROP. A MACROP is just like a regular operator except that it consists of a sequence of actions, not just a single one. A MACROP's preconditions are the initial conditions of the problem just solved, and its postconditions correspond to the goal just achieved. In its simplest form, the caching of previously computed plans is similar to rote learning.

Suppose we are given an initial blocks world situation in which ON(C, B) and ON(A, Table) are both true. STRIPS can achieve the goal ON(A, B) by devising a plan with the four steps UNSTACK(C, B), PUTDOWN(C), PICKUP(A), STA•K(A, B). STRIPS now builds a MACROP with preconditions ON(C, B), ON(A, Table) and postconditions ON(C, Table), ON(A, B). The body of the MACROP consists of the four

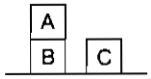
steps just mentioned. In future planning, STRIPS is free to use this complex macro-operator just as it would use any other operator.

But rarely will STRIPS see the exact same problem twice. New problems will differ from previous problems. We would still like the problem solver to make efficient use of the knowledge it gained from its previous experiences. By *generalizing* MACROPs before storing them, STRIPS is able to accomplish this. The simplest idea for generalization is to replace all of the constants in the macro-operator by variables. Instead of storing the MACROP described in the previous paragraph, STRIPS can generalize the plan to consist of the steps UNSTACK(x_1, x_2), PUTDOWN(x_1), PICKUP(x_3), STACK(x_3 x_2), where x_1, x_2, and x_3 are variables. This plan can then be stored with preconditions ON(x_1, x_2), ON(x_3, Table) and postconditionsON(x_1, Table), ON(x_2, x_3). Such a MACROP can now apply in a variety of situations.

Generalization is not so easy, however. Sometimes constants must retain their specific values. Suppose our domain included an operator called STACK-ON-B(x), with preconditions that both x and B be clear, and with postcondition ON(x, B). Consider the same problem as above:

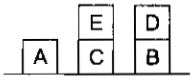


start: ON(C, B)

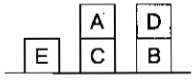


goal: ON(A, B)

STRIPS might come up with the plan UNSTACK(C, B), PUTDOWN(C), STACK-ON-B(A). Let's generalize this plan and store it as a MACROP. The precondition becomes ON(x_3, x_2), the postcondition becomes ON(x_1, x_2), and the plan itself becomes UNSTACK(x_3, x_2), PUTDOWN(x_3), STACK-ON-B(x_1). Now, suppose we encounter a slightly different problem:



start: ON(E, C)
ON(D, B)



goal: ON(A, C)

The generalized MACROP we just stored seems well-suited to solving this problem if we let x_1 = A, x_2 = C, and x_3 = E. Its preconditions are satisfied, so we construct the plan UNSTACK(E, C), PUTDOWN(E), STACK-ON-B(A). But this plan does not work. The problem is that the postcondition of the MACROP is overgeneralized. This operation is only useful for stacking blocks onto B, which is not what we need in this new example. In this case, this difficulty will be discovered when the last step is attempted. Although we cleared C, which is where we wanted to put A, we failed to clear B, which is were the MACROP is going to try to put it. Since B is not clear, STACK-ON-B cannot be executed. If B had happened to be clear, the MACROP would have executed to completion, hut it would not have accomplished the stated goal.

In reality, STRIPS uses a more complex generalization procedure. First, all constants are replaced by variables. Then, for each operator in the parameterized plan, STRIPS revaluates its preconditions. In our example, the preconditions of steps 1 and 2 are satisfied, but the only way to ensure that B is clear for step 3 is to assume that block x_2, which was cleared by the UNSTACK operator, is actually block B. Through "re-proving" that the generalized plan works, STRIPS locates constraints of this kind.

More recent work on macro-operators appears in Korf [1985b]. It turns out that the set of problems for which macro-operators are critical are exactly those problems with *nonserializable subgoals*. Nonserializability means that working on one subgoal will necessarily interfere with the previous solution to another subgoal. Recall that we discussed such problems in connection with nonlinear planning (Section 13.5). Macro-operators can be useful in such cases, since one macro-operator can produce a small global change in the world, even though the individual operators that make it up produce many undesirable local changes.

For example, consider the 8-puzzle. Once a program has correctly placed the first four tiles, it is difficult to place the fifth tile without disturbing the first four. Because disturbing previously solved subgoals is detected as a bad thing by heuristic scoring functions, it is strongly resisted. For many problems, including the 8-puzzle and Rubik's cube, weak methods based on heuristic scoring are therefore insufficient. Hence, we either need domain-specific knowledge, or else a new weak method. Fortunately, we can *learn* the domain-specific knowledge we need in the form of macro-operators. Thus, macro-operators can be viewed as a weak method for learning. In the 8-puzzle, for example, we might have a macro—a complex, prestored sequence of operators—for placing the fifth tile without disturbing any of the first four tiles externally (although in fact they are disturbed within the macro itself). Korf [1985b] gives an algorithm for learning a complete set of macro-operators. This approach contrasts with STRIPS, which learned its MACROPs gradually, from experience. Korf's algorithm runs in time proportional to the time it takes to solve a single problem without macro-operators.

17.4.3 Learning by Chunking

Chunking is a process similar in flavor to macro-operators. The idea of chunking comes from the psychological literature on memory and problem solving. Its computational basis is in production systems, of the type studied in Chapter 6. Recall that in that chapter we described the SOAR system and discussed its use of control knowledge. SOAR also exploits chunking [Laird *et al.,* 1986] so that its performance can increase with experience. In fact, the designers of SOAR hypothesize that chunking is a universal learning method, i.e., it can account for all types of learning in intelligent systems.

SOAR solves problems by firing productions, which are stored in long-term memory. Some of those firings turn out to be more useful than others. When SOAR detects a useful sequence of production firings, it creates a chunk, which is essentially a large production that does the work of an entire sequence of smaller ones. As in MACROPs, chunks are generalized before they are stored.

Recall from Section 6.5 that SOAR is a uniform processing architecture. Problems like choosing which subgoals to tackle and which operators to try (i.e., search control problems) are solved with the same mechanisms as problems in the original problem space. Because tbe problem-solving is uniform, chunking can be used to learn general search control knowledge in addition to operator sequences. For example, if SOAR tries several different operators, but only one leads to a useful path in the search space, then SOAR builds productions that help it choose operators more wisely in the future.

SOAR has used chunking to replicate the macro-operator results described in the last section. In solving the 8-puzzle, for example, SOAR learns how to place a given tile without permanently disturbing the previously placed tiles. Given the way that SOAR learns, several chunks may encode a single macro-operator, and one chunk may participate in a number of macro sequences. Chunks are generally applicable toward any goal state. Tbis contrasts with macro tables, which are structured toward reaching a particular goal state from any initial state. Also, chunking emphasizes how learning can occur during problem-solving, while macro tables are usually built during a preprocessing stage. As a result, SOAR is able to learn within trials as well as across trials. Chunks learned during the initial stages of solving a problem are applicable in the later stages of the same problem-solving episode. After a solution is found, the chunks remain in memory, ready-for-use in the next problem.

The price that SOAR pays for this generality and flexibility is speed. At present, chunking is inadequate for duplicating the contents of large, directly-computed macro-operator tables.

17.4.4 The Utility Problem

PRODIGY [Minton *et al.,* 1989], which we described in Section 6.5, also acquires control knowledge automatically. PRODIGY employs several learning mechanisms. One mechanism uses *explanation-based learning* (EBL), a learning method we discuss in Section 17.6. PRODIGY can examine a trace of its own problem-solving behavior and try to explain why certain paths failed. The program; uses those explanations

to fo[illegible] primarily from examples of successful problem solving, PRODIGY also learns from its failures.

A major contribution of the work on EBL in PRODIGY [Minton, 1988] was the identification of the *utility problem* in learning systems. While new search control knowledge can be of great benefit in solving future problems efficiently, there are also some drawbacks. The learned control rules can take up large amounts of memory and the search program must take the time to consider each rule at each step during problem solving. Considering a control rule amounts to seeing if its postconditions are desirable and seeing if its preconditions are satisfied This is a time-consuming process. So while learned rules may reduce problem-solving time by directing the search more carefully, they may also increase problem-solving time by forcing the problem solver to consider them. If we only want to minimize the number of node expansions in the search space, then the more control rules we learn, the better. But if we want to minimize the total CPU time required to solve a problem, we must consider this trade-off.

PRODIGY maintains a utility measure for each control rule. This measure takes into account the average savings provided by the rule, the frequency of its application, and the cost of matching it. If a proposed rule has a negative utility, it is discarded (or "forgotten"). If not, it is placed in long-term memory with the other rules. It is then monitored during subsequent problem solving. If its utility falls, the rule is discarded. Empirical experiments have demonstrated the effectiveness of keeping only those control rules with high utility. Utility considerations apply to a wide range of learning systems. For example, for a discussion of how to deal with large, expensive chunks in SOAR, see Tambe and Rosenbloom [1989].

17.5 LEARNING FROM EXAMPLES: INDUCTION

Classification is the process of assigning to a particular input, the name of a class to which it belongs. The classes from which the classification procedure can choose can be described in a variety of ways. Their definition will depend on the use to which they will be put.

Classification is an important component of many problem-solving tasks. In its simplest form, it is presented as a straightforward recognition task. An example of this is the question "What letter of the alphabet is this?" But often classification is embedded inside another operation. To see how this can happen, consider a problem-solving system that contains the following production rule:

```
If:   the current goal is to get from place A to place B, and
      there is a WALL separating the two places
then: look for a DOORWAY in the WALL and go through it.
```

To use this rule successfully, the system's matching routine must be able to identify an object as a wall. Without this, the rule can never be invoked. Then, to apply the rule, the system must be able to recognize a doorway.

Before classification can be done, the classes it will use must be defined. This can be done in a variety of ways, including:

- Isolate a set of features that are relevant to the task domain. Define each class by a weighted sum of values of these features. Each class is then defined by a scoring function that looks very similar to the scoring functions often used in other situations, such as game playing. Such a function has the form:

  $$c_1t_1 + c_2t_2 + c_3t_3 + \ldots$$

  Each t corresponds to a value of a relevant parameter, and each c represents the weight to be attached to the corresponding t. Negative weights can be used to indicate features whose presence usually constitutes negative evidence for a given class.

For example, if the task is weather prediction, the parameters can be such measurements as rainfall and location of cold fronts. Different functions can be written to combine these parameters to predict sunny, cloudy, rainy, or snowy weather.

- Isolate a set of features that are relevant to the task domain. Define each class as a structure composed of those features.
  For example, if the task is to identify animals, the body of each type of animal can be stored as a structure, with various features representing such things as color, length of neck, and feathers.

There are advantages and disadvantages to each of these general approaches. The statistical approach taken by the first scheme presented here is often more efficient than the structural approach taken by the second. But the second is more flexible and more extensible.

Regardless of the way that classes are to be described, it is often difficult to construct, by hand, good class definitions. This is particularly true in domains that are not well understood or that change rapidly. Thus the idea of producing a classification program that can evolve its own class definitions is appealing. This task of constructing class definitions is called *concept learning*, or *induction*. The techniques used for this task must, of course, depend on the way that classes (concepts) are described. If classes are described by scoring functions, then concept learning can be done using the technique of coefficient adjustment described in Section 17.4.1. If, however, we want to define classes structurally, some other technique for learning class definitions is necessary. In this section, we present three such techniques.

17.5.1 Winston's Learning Program

Winston [1975] describes an early structural concept learning program. This program operated in a simple blocks world domain. Its goal was to construct representations of the definitions of concepts in the blocks domain. For example, it learned the concepts *House, Tent,* and *Arch* shown in Fig. 17.2. The figure also shows an example of a near miss for each concept. A *near miss* is an object that is not an instance of the concept in question but that is very similar to such instances.

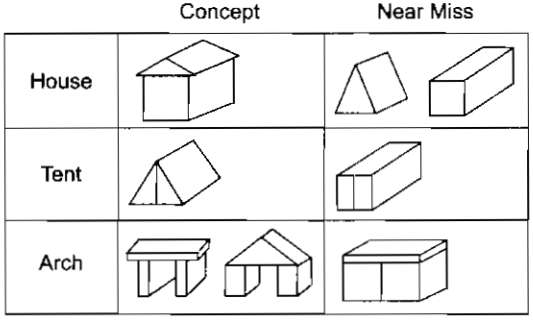


Fig. 17.2 *Some Blocks World Concepts*

The program started with a line drawing of a blocks world structure. It used procedures such as the one described in Section 14.3 to analyze the drawing and construct a semantic net representation of the structural description of the object(s) .This structural description was then provided as input to the learning program. An example of such a structural description for the *House* of Fig. 17.2 is shown in Fig. 17.3(a). Node A represents the entire structure, which is composed of two parts: node B, a *Wedge*, and node C, a *Brick*. Figures 17.3(b) and 17.3(c) show descriptions of the two *Arch* structures of Fig. 17.2. These descriptions are identical except for the types of the objects on the top; one is a *Brick* while the other is a *Wedge*. Notice that the two supporting objects are related not only by *left-of* and *right-of* links, but also by a *does-not-marry* link, which says that the two objects do not *marry*. Two objects *marry* if they have faces that touch and they have a common edge. The *marry* relation is critical in the definition of an *Arch*. It is the difference between the first arch structure and the near miss arch structure shown in Fig. 17.2.

The basic approach that Winston's program took to the problem of concept formation can be described as follows:

1. Begin with a structural description of one known instance of the concept. Call that description the concept definition.

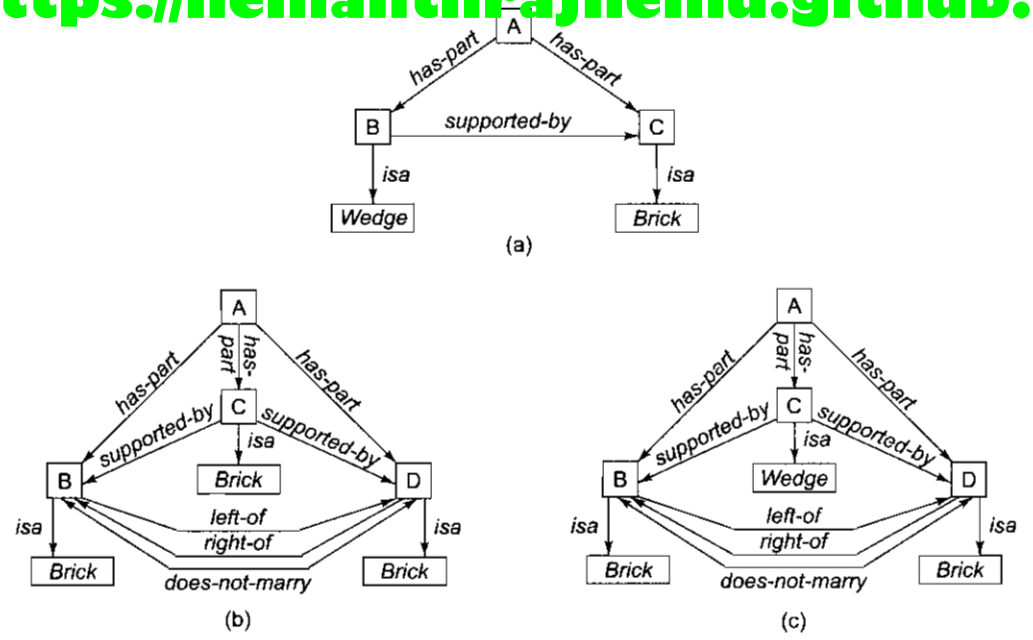


Fig. 17.3 *Structural Descriptions*

2. Examine descriptions of other known instances of the concept. Generalize the definition to include them.
3. Examine descriptions of near misses of the concept. Restrict the definition to exclude these.

Steps 2 and 3 of this procedure can be interleaved.

Steps 2 and 3 of this procedure rely heavily on a comparison process by which similarities and differences between structures can be detected. This process must function in much the same way as does any other matching process, such as one to determine whether a given production rule can be applied to a particular problem state. Because differences as well as similarities must be found, the procedure must perform not just literal but also approximate matching. The output of the comparison procedure is a skeleton structure describing the commonalities between the two input structures. It is annotated with a set of comparison notes that describe specific similarities and differences between the inputs.

To see how this approach works, we trace it through the process of learning what an arch is. Suppose that the arch description of Fig. 17.3(b) is presented first. It then becomes the definition of the concept *Arch*. Then suppose that the arch description of Fig. 17.3(c) is presented. The comparison routine will return a structure similar to the two input structures except that it will note that the objects represented by the nodes labeled C are not identical. This structure is shown as Fig. 17.4. The *c-note* link from node C describes

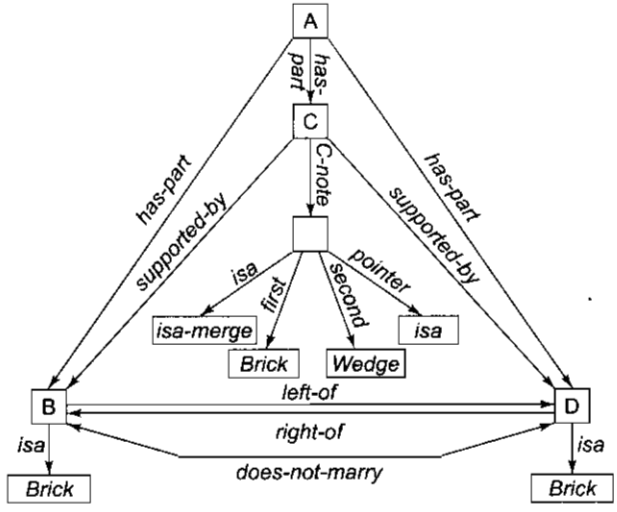


Fig. 17.4 *The Comparison of Two Arches*

the difference found by the comparison routine. It notes that the difference occurred in the *isa* link, and that in the first structure the *isa* link pointed to *Brick*, and in the second it pointed to *Wedge*. It also notes that if we were to follow *isa* links from *Brick* and *Wedge*, these links would eventually merge. At this point, a new description of the concept *Arch* can be generated. This description could say simply that node C must be either a *Brick* or a *Wedge*. But since this particular disjunction has no previously known significance, it is probably better to trace up the *isa* hierarchies of *Brick* and *Wedge* until they merge. Assuming that that happens at the node *Object*, the *Arch* definition shown in Fig. 17.5 can be built.

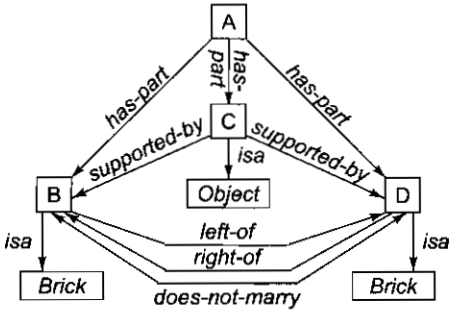


Fig. 17.5 *The Arch Description after Two Examples*

Next, suppose that the near miss arch shown in Fig. 17.2 is presented. This time, the comparison routine will note that the only difference between the current definition and the near miss is in the *does-not-marry* link between nodes *B* and *D*. But since this is a near miss, we do not want to broaden the definition to include it. Instead, we want to restrict the definition so that it is specifically excluded. To do this, we modify the link *does-not-marry*, which may simply be recording something that has happened by chance to be true of the small number of examples that have been presented. It must now say *must-not-marry*. The *Arch* description at this point is shown in Fig. 17.6. Actually, *must-not-marry* should not be a completely new link. There must be some structure among link types to reflect the relationship between *marry*, *does-not-marry*, and *must-not-marry*.

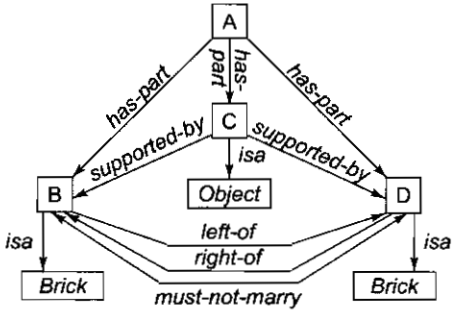


Fig. 17.6 *The Arch Description after a Near Miss*

Notice how the problem-solving and knowledge representation techniques we covered in earlier chapters are brought to bear on the problem of learning. Semantic networks were used to describe block structures, and an *isa* hierarchy was used to de-scribe relationships among already known objects. A matching process was used to detect similarities and differences between structures, and hill climbing allowed the program to evolve a more and more accurate concept definition.

This approach to structural concept learning is not without its problems. One major problem is that a teacher must guide the learning program through a carefully chosen sequence of examples. In the next section, we explore a learning technique that is insensitive to the order in which examples are presented.

17.5.2 Version Spaces

Mitchell [1977; 1978] describes another approach to concept learning called *version spaces*. The goal is the same: to produce a description that is consistent with all positive examples but no negative examples in the training set. But while Winston's system did this by evolving a single concept description, version spaces work by maintaining a *set* of possible descriptions and evolving that set as new examples and near misses are presented. As in the previous section, we need some sort of representation language for examples so that we can describe exactly what the system sees in an example. For now we assume a simple frame-based language; although version spaces can be constructed for more general representation languages. Consider Fig. 17.7, a frame representing an individual car.

origin :	*Japan*
manufacturer :	*Honda*
color :	*Blue*
decade :	*1970*
type :	*Economy*

Fig. 17.7 *An Example of the Concept Car*

Now, suppose that each slot may contain only the discrete values shown in Fig. 17.8. The choice of features and values is called the *bias* of the learning system. By being embedded in a particular program and by using particular representations, every learning system is biased, because it learns some things more easily than others. In our example, the bias is fairly simple — e.g., we can learn concepts that have to do with car manufacturers, but not car owners. In more complex systems, the bias is less obvious. A clear statement of the bias of a learning system is very important to its evaluation.

origin	$\in$	{*Japan, USA, Britain, Germany, Italy*}
manufacturer	$\in$	{*Honda, Toyota, Ford, Chrysler, Jaguar, BMW, Fiat*}
color	$\in$	{*Blue, Green, Red, White*}
decade	$\in$	{1950, 1960, 1970, 1980, 1990, 2000}
type	$\in$	{*Economy, Luxury, Sports*}

Fig. 17.8 *Representation Language for Cars*

Concept descriptions, as well as training examples, can be stated in terms of these slots and values. For example, the concept "Japanese economy car" can be represented as in Fig. 17.9. The names x_1, x_2, and x_3 are variables. The presence of x_2, for example, indicates that the color of a car is not relevant to whether the car is a Japanese economy car. Now the learning problem is: Given a representation language such as in Fig. 17.8, and given positive and negative training examples such as those in Fig. 17.7, how can we produce a concept description such as that in Fig. 17.9 that is consistent with all the training examples?

origin :	*Japan*
manufacturer :	x_1
color :	x_2
decade :	x_3
type :	*Economy*

Fig, 17.9 *The Concept "Japanese economy car"*

Before we proceed to the version space algorithm, we should make some observations about the representation. Some descriptions are more general than others. For example, the description in Fig. 17.9 is more general than the one in Fig. 17.7. In fact, the representation language defines a partial ordering of descriptions. A portion of that partial ordering is shown in Fig. 17.10.

The entire partial ordering is called the *concept space,* and can be depicted as in Fig. 17.11. At the top of the concept space is the null description, consisting only of variables, and at the bottom are all the possible training instances, which contain no variables. Before we receive any training examples, we know that the target concept lies somewhere in the concept space. For example, if every possible description is an instance of the intended concept, then the null description is the concept definition since it matches everything. On the other hand, if the target concept includes only a single example, then one of the descriptions at the bottom of the concept space is the desired concept definition. Most target concepts, of course, lie somewhere in between these two extremes.

As we process training examples, we want to refine our notion of where the target concept might lie. Our current hypothesis can be represented as a subset of the concept space called the *version space.* The version space is the largest collection of descriptions that is consistent with all the training examples seen so far.

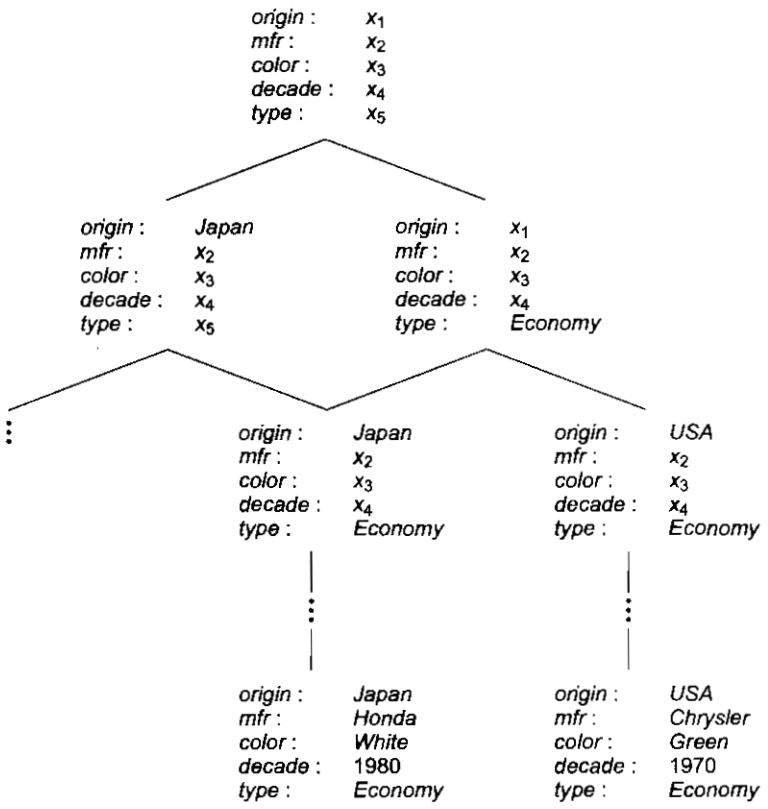


Fig. 17.10 *Partial Ordering of Concepts Specified by the Representation Language*

How can we represent the version space? The version space is simply a set of descriptions, so an initial idea is to keep an explicit list of those descriptions. Unfortunately, the number of descriptions in the concept space is exponential in the number of features and values. So enumerating them is prohibitive. However, it turns out that the version space has a concise representation. It consists of two subsets of the concept space. One subset, called G contains the most *general* descriptions consistent with the training examples seen so far; the other subset, called S, contains the most *specific* descriptions consistent with the training examples. The version space is the set of all descriptions that lie between some element of G and some element of S in the partial order of the concept space.

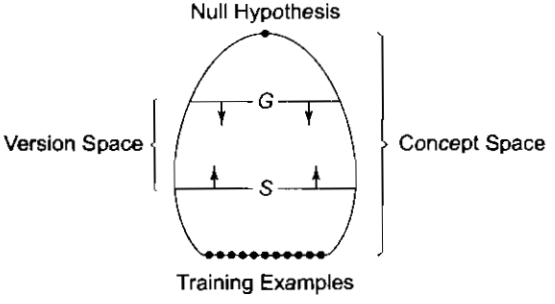


Fig. 17.11 *Concept and Version Spaces*

This representation of the version space is not only efficient for storage, but also for modification. Intuitively, each time we receive a positive training example, we want to make the S set more general. Negative training examples serve to make the G set more specific. If the S and G sets converge, our range of hypotheses will narrow to a single concept description. The algorithm for narrowing the version space is called the *candidate elimination algorithm.*

Algorithm: Candidate Elimination

Given: A representation language and a set of positive and negative examples expressed in that language.

Compute: A concept description that is consistent with all the positive examples and none of the negative examples.

1. Initialize G to contain one element: the null description (all features are variables).
2. Initialize S to contain one element: the first positive example.
3. Accept a new training example.
   If it is a *positive example,* first remove from G any descriptions that do not cover the example. Then, update the S set to contain the most specific set of descriptions in the version space that cover the example and the current elements of the S set.
   That is, generalize the elements of S as little as possible so that they cover the new training example.
   If it is a *negative* example, first remove from S any descriptions that cover the example. Then, update the G set to contain the most general set of descriptions in the version space that *do not* cover the example. That is, specialize the elements of G as little as possible so that the negative example is no longer covered by any of the elements of G.
4. If S and G are both singleton sets, then if they are identical, output their value and halt. If they are both singleton sets but they are different, then the training cases were inconsistent. Output this result and halt. Otherwise, go to step 3.

Let us trace the operation of the candidate elimination algorithm. Suppose we want to learn the concept of "Japanese economy car" from the examples in Fig. 17.12. G and S both start out as singleton sets. G contains the null description (see Fig. 17.11), and S contains the first positive training example. The version space now contains all descriptions that are consistent with this first example:[2]

origin:	*Japan*	*origin:*	*Japan*	*origin:*	*Japan*	*origin:*	*USA*	*origin:*	*Japan*
mfr:	*Honda*	*mfr:*	*Toyota*	*mfr:*	*Toyota*	*mfr:*	*Chrysler*	*mfr:*	*Honda*
color:	*Blue*	*color:*	*Green*	*color:*	*Blue*	*color:*	*Red*	*color:*	*White*
decade:	1980	*decade:*	1970	*decade:*	1990	*decade:*	1980	*decade:*	1980
type:	*Economy*	*type:*	*Sports*	*type:*	*Economy*	*type:*	*Economy*	*type:*	*Economy*
(+)		(−)		(+)		(−)		(+)	

Fig. 17.12 *Positive and Negotive Examples of the Concept "Japanese economy car"*

$G = \{(x_1, x_2, x_3, x_4, x_5)\}$

$S = \{(Japan, Honda, Blue, 1980, Economy)\}$

Now we are ready to process the second example. The G set must be specialized in such a way that the negative example is no longer in the version space. In our representation language, specialization involves replacing variables with constants. (Note: The G set must be specialized only to descriptions that are *within* the current version space, not outside of it.) Here are the available specializations:

[2] To make this example concise, we skip slot names in the descriptions. We just list slot values in the order in which the slots have been shown in the preceding figures.

$$G = \{(x_1, Honda, x_3, x_4, x_5), (x_1, x_2, Blue, x_4, x_5), (x_1, x_2, x_3, 1980, x_5), (x_1, x_2, x_3, x_4, Economy)\}$$

The S set is unaffected by the negative example. Now we come to the third example, a positive one. The first order of business is to remove from the G set any descriptions that are inconsistent with the positive example. Our new G set is:

$$G = \{(x_1, x_2, Blue, x_4, x_5), (x_1, x_2, x_3, x_4, Economy)\}$$

We must now generalize the S set to include the new example. This involves replacing constants with variables. Here is the new S set:

$$S = \{(Japan, x_2, Blue, x_4, Economy)\}$$

At this point, the S and G sets specify a version space (a space of candidate descriptions) that can be translated roughly into English as: "The target concept may be as specific as 'Japanese, blue economy car,' or as general as either 'blue car' or 'economy car.'"

Next, we get another negative example, a car whose *origin* is *USA*. The S set is unaffected, but the G set must be specialized to avoid covering the new example. The new G set is:

$$G = \{(Japan, x_2, Blue, x_4, x_5), (Japan, x_2, x_3, x_4, Economy)\}$$

We now know that the car must be Japanese, because *all* of the descriptions in the version space contain *Japan* as *origin*.[3] Our final example is a positive one. We first remove from the G set any descriptions that are inconsistent with it, leaving:

$$G = \{(Japan, x_2, x_3, x_4, Economy)\}$$

We then generalize the S set to include the new example:

$$S = \{(Japan, x_2, x_3, x_4, Economy)\}$$

S and G are both singletons, so the algorithm has converged on the target concept. No more examples are needed.

There are several things to note about the candidate elimination algorithm. First, it is a *least-commitment* algorithm. The version space is pruned as little as possible at each step. Thus, even if all the positive training examples are Japanese cars, the algorithm will not reject the possibility that the target concept may include cars of other origin—until it receives a negative example that forces the rejection. This means that if the training data are sparse, the S and G sets may never converge to a single description; the system may learn only partially specified concepts. Second, the algorithm involves exhaustive, breadth-first search through the version space. We can see this in the algorithm for updating the G set. Contrast this with the depth-first behavior of Winston's learning program. Third, in our simple representation language, the S set always contains exactly one element, because any two positive examples always have exactly one generalization. Other representation languages may not share this property.

[3] It could be the case that our target concept is "not Chrysler," but we will ignore this possibility because our representation language is not powerful enough to express negation and disjunction.

The version space approach can be applied to a wide variety of learning tasks and representation languages. The algorithm above can be extended to handle continuously valued features and hierarchical knowledge (see Exercises). However, version spaces have several deficiencies. One is the large space requirements of the exhaustive, breadth-first search mentioned above. Another is that inconsistent data, also called *noise,* can cause the candidate elimination algorithm to prune the target concept from the version space prematurely. In the car example above, if the third training instance had been mislabeled (–) instead of (+), the target concept of "Japanese economy car" would never be reached. Also, given enough erroneous negative examples, the G set can be specialized so far that the version space becomes empty. In that case, the algorithm concludes that *no* concept fits the training examples.

One solution to this problem [Mitchell, 1978] is to maintain several G and S sets. One G set is consistent with all the training instances, another is consistent with all but one, another with all but two, etc. (and the same for the S set). When an inconsistency arises, the algorithm switches to G and S sets that are consistent with most, but not all, of the training examples. Maintaining multiple version spaces can be costly, however, and the S and G sets are typically very large. If we assume *bounded inconsistency,* i.e., that instances close to the target concept boundary are the most likely to be misclassified, then more efficient solutions are possible. Hirsh [1990] presents an algorithm that runs as follows. For each instance, we form a version space consistent with that instance plus other nearby instances (for some suitable definition of nearby). This version space is then intersected with the one created for all previous instances. We keep accepting instances until the version space is reduced to a small set of candidate concept descriptions. (Because of inconsistency, it is unlikely that the version spaec will converge to a singleton.) We then match each of the concept descriptions against the entire data set, and choose the one that classifies the instances most accurately.

Another problem with the candidate elimination algorithm is the learning of disjunctive concepts. Suppose we wanted to learn the concept of "European car," which, in our representation, means either a German, British, or Italian car. Given positive examples of each, the candidate elimination algorithm will generalize to cars of any *origin.* Given such a generalization, a negative instance (say, a Japanese car) will only cause an inconsistency of the type mentioned above.

Of course, we could simply extend the representation language to include disjunctions. Thus, the concept space would hold descriptions such as "Blue car of German or British origin" and "Italian sports car or German luxury car." This approach has two drawbacks. First, the concept space becomes much larger and specialization becomes intractable. Second, generalization can easily degenerate to the point where the S set contains simply one large disjunction of all positive instances. We must somehow force generalization while allowing for the introduction of disjunctive descriptions. Mitchell [1978] gives an iterative approach that involves several passes through the training data. On each pass, the algorithm builds a concept that covers the largest number of posiiive training instances without covering any negative training instances. At the end of the pass, the positive training instances covered by the new concept are removed from the training set, and the new concept then becomes one disjunct in the eventual disjunctive concept description. When all positive training instances have been removed, we are left with a disjunctive concept that covers all of them without covering any negative instances.

There are a number of other complexities, including the way in which features interact with one another. For example, if the *origin* of a car is *Japan,* then the *manufacturer* cannot be *Chrysler.* The version space algorithm as described above makes no use of such information. Also in our example, it would be more natural to replace the *decade* slot with a continuously valued *year* field. We would have to change our procedures for updating the S and G sets to account for this kind of numerical data.

17.5.3 Decision Trees

A third approach to concept learning is the induction of *decision trees*, as exemplified by the ID3 program of Quinlan [1986]. ID3 uses a tree representation for concepts, such as the one shown in Fig. 17.13. To classify a particular input, we start at the top of the tree and answer questions until we reach a leaf, where the classification is stored. Fig. 17.13 represents the familiar concept "Japanese economy car." ID3 is a program that builds decision trees automatically, given positive and negative instances of a concept.[4]

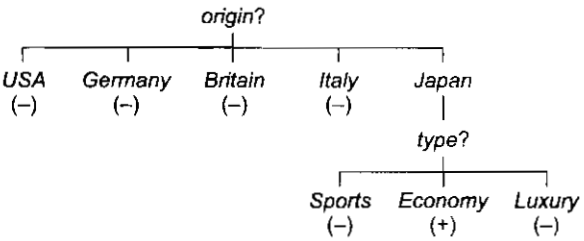


Fig. 17.13 *A Decision Tree*

ID3 uses an iterative method to build up decision trees, preferring simple trees over complex ones, on the theory that simple trees are more accurate classifiers of future inputs. It begins by choosing a random subset of the training examples. This subset is called the *window*. The algorithm builds a decision tree that correctly classifies all examples in the window. The tree is then tested on the training examples outside the window. If all the examples are classified correctly, the algorithm halts. Otherwise, it adds a number of training examples to the window and the process repeats. Empirical evidence indicates that the iterative strategy is more efficient than considering the whole training set at once.

So how does ID3 actually construct decision trees? Building a node means choosing some attribute to test. At a given point in the tree, some attributes will yield more information than others. For example, testing the attribute *color* is useless if the color of a car does not help us to classify it correctly. Ideally, an attribute will separate training instances into subsets whose members share a common label (e.g., positive or negative). In that case, branching is terminated, and the leaf nodes are labeled.

There are many variations on this basic algorithm. For example, when we add a test that has more than two branches, it is possible that one branch has no corresponding training instances. In that case, we can either leave the node unlabeled, or we can attempt to guess a label based on statistical properties of the set of instances being tested at that point in the tree. Noisy input is another issue. One way of handling noisy input is to avoid building new branches if the information gained is very slight. In other words, we do not want to overcomplicate the tree to account for isolated noisy instances. Another source of uncertainty is that attribute values may be unknown. For example a patient's medical record may be incomplete. One solution is to guess the correct branch to take; another solution is to build special "unknown" branches at each node during learning.

When the concept space is very large, decision tree learning algorithms run more quickly than ther version space cousins. Also, disjunction is more straightforward. For example, we can easily modify Fig. 17.13 to represent the disjunctive concept "American car or Japanese economy car," simply by changing one of the negative (—) leaf labels to positive (+). One drawback to the ID3 approach is that large, complex decision trees can be difficult for humans to understand, and so a decision tree system may have a hard time explaining the reasons for its classifications.

17.6 EXPLANATION-BASED LEARNING

The previous section illustrated how we can induce concept descriptions from positive and negative examples. Learning complex concepts using these procedures typically requires a substantial number of training instances.

[4] Actually, the decision tree representation is more general: Leaves can denote any of a number of classes, not just positive and negative.

But people seem to be able to learn quite a bit from single examples. Consider a chess player who, as Black, has reached the position shown in Fig. 17.14. The position is called a "fork" because the white knight attacks both the black king and the black queen. Black must move the king, thereby leaving the queen open to capture. From this single experience, Black is able to learn quite a bit about the fork trap: the idea is that if any piece x attacks both the opponent's king and another piece y, then piece y will be lost. We don't need to see dozens of positive and negative examples of fork positions in order to draw these conclusions. From just one experience, we can learn to avoid this trap in the future and perhaps to use it to our own advantage.

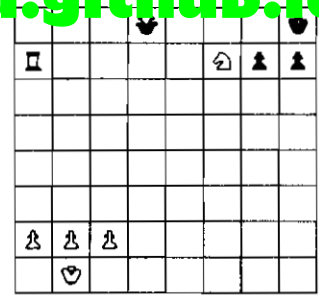


Fig. 17.14 *A Fork Position in Chess*

What makes such single-example learning possible? The answer, not surprisingly, is knowledge. The chess player has plenty of domain-specific knowledge that can be brought to bear, including the rules of chess and any previously acquired strategies. That knowledge can be used to identify the critical aspects of the training example. In the case of the fork, we know that the double simultaneous attack is important while the precise position and type of the attacking piece is not.

Much of the recent work in machine learning has moved away from the empirical, data-intensive approach described in the last section toward this more analytical, knowledge-intensive approach. A number of independent studies led to the characterization of this approach as *explanation-based learning*. An EBL system attempts to learn from a single example x by explaining why x is an example of the target concept. The explanation is then generalized, and the system's performance is improved through the availability of this knowledge.

Mitchell *et al.* [1986] and DeJong and Mooney [1986] both describe general frameworks for EBL programs and give general learning algorithms. We can think of EBL programs as accepting the following as input:

- *A Training Example*—What the learning program "sees" in the world, e.g., the car of Fig. 17.7
- *A Goal Concept*—A high-level description of what the program is supposed to learn
- *An Operationally Criterion*—A description of which concepts are usable
- *A Domain Theory*—A set of rules that describe relationships between objects and actions in a domain

From this, EBL computes a *generalization* of the training example that is sufficient to describe the goal concept, and also satisfies the operationality criterion.

Let's look more closely at this specification. The training example is a familiar input—it is the same thing as the example in the version space algorithm. The goal concept is also familiar, but in previous sections, we have viewed the goal concept as an output of the program, not an input. The assumption here is that the goal concept is not operational, just like the high-level card-playing advice described in Section 17.3. An EBL program seeks to operationalize the goal concept by expressing it in terms that a problem-solving program can understand. These terms are given by the operatioriality criterion. In the chess example, the goal concept might be something like "bad position for Black," and, the operationalized concept would be a generalized description of Situations similar to the training example, given in terms of pieces and their relative positions. The last input to an EBL program is a domain theory, in our case, the rules of chess. Without such knowledge, it is impossible to come up with a correct generalization of the training example.

Explanation-based generalization (EBG) is an algorithm for EBL described in Mitchell *et al.* [1986]. It has two steps: (1) explain and (2) generalize. During the first step, the domain theory is used to prune away all the unimportant aspects of the training example with respect to the goal concept. What is left is an *explanation* of why the training example is an instance of the goal concept. This explanation is expressed in terms that satisfy the operationality criterion. The next step is to generalize the explanation as far as possible while still

describing the goal concept. Following our chess example, the first EBL step chooses to ignore White's pawns, king, and rook, and constructs an explanation consisting of White's knight, Black's king, and Black's queen, each in their specific positions. Operationality is ensured: all chess-playing programs understand the basic concepts of piece and position. Next, the explanation is generalized. Using domain knowledge, we find that moving the pieces to a different part of the board is still bad for Black. We can also determine that other pieces besides knights and queens can participate in fork attacks.

In reality, current EBL methods run into difficulties in domains as complex as chess, so we will not pursue this example further. Instead, let's look at a simpler case. Consider the problem of learning the concept *Cup* [Mitchell *et al.*, 1986]. Unlike the arch-learning program of Section 17.5.1, we want to be able to generalize from a single example of a cup. Suppose the example is:

- Training Example:

  $owner(Object23, Ralph) \wedge has\text{-}part(Object23, Concavity12) \wedge$
  $is(Object23, Light) \wedge color(Object23, Brown) \wedge ...$

Clearly, some of the features of *Object23* are more relevant to its being a cup than others. So far in this chapter, we have seen several methods for isolating relevant features. These methods all require many positive and negative examples. In EBL we instead rely on domain knowledge, such as:

- Domain Knowledge:

  $is(x, Light) \wedge has\text{-}part(x, y) \wedge isa(y, Handle) \rightarrow liftable(x)$
  $has\text{-}part(x, y) \wedge isa(y, Bottom) \wedge is(y, Flat) \rightarrow stable(x)$
  $has\text{-}part(x, y) \wedge isa(y, Concavity) \wedge is(y, Upward\text{-}Pointing) \rightarrow open\text{-}vessel(x)$

We also need a goal concept to operationalize:

- Goal Concept: *Cup*

  x is a Cup if x is *liftable*, *stable*, and *open-vessel*.

- Operationality Criterion: Concept definition must be expressed in purely structural terms (e.g., *Light*, *Flat*, etc.).

Given a training example and a functional description, we want to build a general structural description of a cup. The first step is to explain why *Object23* is a cup. We do this by constructing a proof, as shown in Fig. 17.15. Standard theorem-proving techniques can be used to find such a proof. Notice that the proof

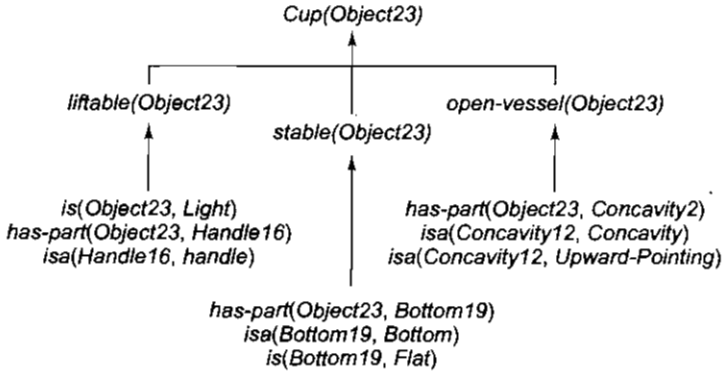


Fig. 17.15 *An Explanation*

isolates the relevant features of the training example; nowhere in the proof do the predicates *owner* and *color* appear. The proof also serves as a basis for a valid generalization. If we gather up all the assumptions and replace constants with variables, we get the following description of a cup:

$$\textit{has-part}(x, y) \wedge \textit{isa}(y, \textit{Concavity}) \wedge \textit{is}(y, \textit{Upward-Pointing}) \wedge$$
$$\textit{has-part}(x, z) \wedge \textit{isa}(z, \textit{Bottom}) \wedge \textit{is}(z, \textit{Flat}) \wedge$$
$$\textit{has-part}(x, w) \wedge \textit{isa}(w, \textit{Handle}) \wedge \textit{is}(x, \textit{Light})$$

This definition satisfies the operationality criterion and could be used by a robot to classify objects.

Simply replacing constants by variables worked in this example, but in some cases it is necessary to retain certain constants. To catch these cases, we must reprove the goal. This process, which we saw earlier in our discussion of learning in STRIPS, is called *goal regression.*

As we have seen, EBL depends strongly on a domain theory. Given such a theory, why are examples needed at all? We could have operationalized the goal concept *Cup* without reference to an example, since the domain theory contains all of the requisite information. The answer is that examples help to focus the learning on relevant operationalizations. Without an example cup, EBL is faced with the task of characterizing the entire range of objects that satisfy the goal concept. Most of these objects will never be encountered in the real world, and so the result will be overly general.

Providing a tractable domain theory is a difficult task. There is evidence that humans do not learn with very primitive relations. Instead, they create incomplete and inconsistent domain theories. For example, returning to chess, such a theory might include concepts like "weak pawn structure." Getting EBL to work in ill-structured domain theories is an active area of research (see, e.g., Tadepalli [1989]).

EBL shares many features of all the learning methods described in earlier sections. Like concept learning, EBL begins with a positive example of some concept. As in learning by advice taking, the goal is to operationalize some piece of knowledge. And EBL techniques, like the techniques of chunking and macro-operators, are often used to improve the performance of problem-solving engines. The major difference between EBL and other learning methods is that EBL programs are built to take advantage of domain knowledge. Since learning is just another kind of problem solving, it should come as no surprise that there is leverage to be found in knowledge.

17.7 DISCOVERY

Learning is the process by which one entity acquires knowledge. Usually that knowledge is already possessed by some number of other entities who may serve as teachers. *Discovery* is a restricted form of learning in which one entity acquires knowledge without the help of a teacher.[5] In this section, we look at three types of automated discovery systems.

17.7.1 AM: Theory-Driven Discovery

Discovery is certainly learning. But it is also, perhaps more clearly than other kinds of learning, problem-solving. Suppose that we want to build a program to discover things, for example, in mathematics. We expect that such a program would have to rely heavily on the problem-solving techniques we have discussed. In fact, one such program was written by Lenat [1977; 1982]. It was called AM, and it worked from a few basic concepts of set theory to discover a good deal of standard number theory.

[5] Sometimes, there is no one in the world who has the knowledge we seek. In that case, the kind of action we must take is called *scientific discovery.*

AM exploited a variety of general-purpose AI techniques. It used a frame system to represent mathematical concepts. One of the major activities of AM is to create new concepts and fill in their slots. An example of an AM concept is shown in Fig. 17.16. AM also uses heuristic search, guided by a set of 250 heuristic rules representing hints about activities that are likely to lead to "interesting" discoveries. Examples of the kind of heuristics AM used are shown in Fig. 17.17. Generate-and-test is used to form hypotheses on the basis of a small number of examples and then to test the hypotheses on a larger set to see if they still appear to hold. Finally, an agenda controls the entire discovery process. When the heuristics suggest a task, it is placed on a central agenda, along with the reason that it was suggested and the strength with which it was suggested. AM operates in cycles, each time choosing the most promising task from the agenda and performing it.

name : *Prime-Numbers*
definitions :
    origin : *Number-of-divisors-of*(x) = 2
    predicate-calculus: Prime(x) $\Leftrightarrow (\forall z)(z \mid x \Rightarrow (z = 1 \otimes z = x))$
    iterative : (*for* $x > 1$): For i from 2 to $\sqrt{x}$, $i \not\mid x$
examples : 2, 3, 5, 7, 11, 13, 17
    boundary : 2, 3
    boundary-failures : 0, 1
    failures : 12
generalizations : *Number, numbers with an even number of divisors*
specializations : *Odd primes, prime pairs, prime uniquely addables*
conjecs : *Unique factorization, Goldbach's conjecture, extremes of number-of-divisors-of*
intus : *A metaphor to the effect that primes are the building blocks of all numbers*
analogies :
    Maximally divisible numbers are converse extremes of number-of-divisors-of
    Factor a nonsimple group into simple groups
interest : *Conjectures tying primes to times, to divisors of, to related operations*
worth : 800

Fig. 17.16 *An AM Concept: Prime Number*

- If f is a function from A to B and B is ordered, then consider the elements of A that are mapped into extremal elements of B. Create a new concept representing this subset of A.
- If some (but not most) examples of some concept X are also examples of another concept Y, create a new concept representing the intersection of X and Y.
- If very few examples of a concept X are found, then add to the agenda the task of finding a generalization of X.

Fig. 17.17 *Some AM Heuristics*

In one run, AM discovered the concept of prime numbers. How did it do that? Having stumbled onto the natural numbers, AM explored operations such as addition, multiplication, and their inverses. It created the concept of divisibility and noticed that some numbers had very few divisors. AM has a built-in heuristic that tells it to explore extreme cases. It attempted to list all numbers with zero divisors (finding none), one divisor (finding one: 1), and two divisors. AM was instructed to call the last concept "primes." Before pursuing this concept, AM went on to list numbers with three divisors, such as 49. AM tried to relate this property with other properties of 49, such as its being odd and a perfect square. AM generated other odd numbers and other perfect squares to test its hypotheses. A side effect of determining the equivalence of perfect squares with numbers with three divisors was to boost the "interestingness" rating of the divisor concept. This led; AM to investigate ways in which a number could be broken down into factors. AM then noticed that there was only one way to break a number down into prime factors (known as the Unique Factorization Theorem).

Since breaking down numbers into multiplicative components turned out to be interesting, AM decided, by analogy, to pursue additive components as well. It made several uninteresting conjectures, such as that

every number could be expressed as a sum of 1 's. It also found more interesting phenomena, such as that many numbers were expressible as the sum of two primes. By listing cases, AM determined that all even numbers greater than 2 seemed to have this property. This conjecture, known as Goldbach's Conjecture, is widely believed to be true, but a proof of it has yet to be found in mathematics.

AM contains a great many general-purpose heuristics such as the ones it used in this example. Often different heuristics point in the same place. For example, while AM discovered prime numbers using a heuristic that involved looking at extreme cases, another way to derive prime numbers is to use the following two rules:

- If there is a strong analogy between A and B but there is a conjecture about A that does not hold for all elements of B, define a new concept that includes the elements of B for which it does hold.
- If there is a set whose complement is much rarer than itself, then create a new concept representing the complement.

There is a strong analogy between addition and multiplication of natural numbers. But that analogy breaks down when we observe that all natural numbers greater than 1 can be expressed as the sum of two smaller natural numbers (excluding the identity). This is not true for multiplication. So the first heuristic described above suggests the creation of a new concept representing the set of composite numbers. Then the second heuristic suggests creating a concept representing the complement of that, namely the set of prime numbers.

Two major questions came out of the work on AM. One question was: "Why was AM ever turned off?" That is, why didn't AM simply keep discovering new interesting facts about numbers, possibly facts unknown to human mathematics? Lenat [1983b] contends that AM's performance was limited by the static nature of its heuristics. As the program progressed, the concepts with which it was working evolved away from the initial ones, while the heuristics that were available to work on those concepts stayed the same. To remedy this problem, it was suggested that heuristics be treated as full-fledged concepts that could be created and modified by the same sorts of processes (such as generalization, specialization, and analogy) as are concepts in the task domain. In other words, AM would run in discovery mode in the domain of "Heuretics," the study of heuristics themselves, as well as in the domain of number theory. An extension of AM called EURISKO [Lenat, 1983a] was designed with this goal in mind.

The other question was: "Why did AM work as well as it did?" One source of power for AM was its huge collection of heuristics about what constitute interesting things. But AM had another less obvious source of power, namely, the natural relationship between number theoretical concepts and their compact representations in AM [Lenat and Brown, 1983]. AM worked by syntactically mutating old concept definitions— stored essentially as short LISP programs—in the hopes of finding new, interesting concepts. It turns out that a mutation in a small LISP program very likely results in another well-formed, meaningful LISP program. This accounts for AM's ability to generate so many novel concepts. But while humans interpret AM as exploring number theory, it was actually exploring the space of small LISP programs. AM succeeded in large part because of this intimate relationship between number theory and LISP programs. When AM and EURISKO were applied to other domains, including the study of heuristics themselves, problems arose. Concepts in these domains were larger and more complex than number theory concepts, and the syntax of the representation language no longer closely mirrored the semantics of the domain. As a result, syntactic mutation of a concept definition almost always resulted in an ill-formed or useless concept, severely hampering the discovery procedure.

Perhaps the moral of AM is that learning is a tricky business. We must be careful how we interpret what our AI programs are doing [Ritchie and Hanna, 1984]. AM had an implicit *bias* toward learning concepts in number theory. Only after that bias was explicitly recognized was it possible to understand why AM performed well in one domain and poorly in another.

17.7.2 BACON: Data-Driven Discovery

AM showed how discovery might occur in a theoretical setting. Empirical scientists see things somewhat differently. They are confronted with data from the world and must make "sense of it. They make hypotheses, and in order to validate them, they design and execute experiments. Scientific discovery has inspired a number of computer models. Langley *et al.* [1981 a] present a model of data-driven scientific discovery that has been implemented as a program called BACON, named after Sir Francis Bacon, an early philosopher of science.

BACON begins with a set of variables for a problem. For example, in the study of the behavior of gases, some variables are p, the pressure on the gas, V, the volume of the gas, n, the amount of gas in moles, and T, the temperature of the gas. Physicists have long known a law, called the *ideal gas law*, that relates these variables. BACON is able to derive this law on its own. First, BACON holds the variables n and T constant, performing experiments at different pressures p_1, p_2, and p_3. BACON notices that as the pressure increases, the volume V decreases. Therefore, it creates a theoretical term pV. This term is constant. BACON systematically moves on to vary the other variables. It tries an experiment with different values of T, and finds that pV changes. The two terms are linearly related with an intercept of 0, so BACON creates a new term pV/T. Finally, BACON varies the term n and finds another linear relation between n and pV/T. For all values of n, p, V, and T, $pV/nT = 8.32$. This is, in fact, the ideal gas law. Fig. 17.18 shows BACON's reasoning in a tabular format.

n	T	p	V	pV	pV/T	pV/nT
1	300	100	24.96			
1	300	200	12.48			
1	300	300	8.32	2496		
1	310			2579.2		
1	320			2662.4	8.32	
2	320				16.64	
3	320				24.96	8.32

Fig. 17.18 *BACON Discovering the Ideal Gas Law*

BACON has been used to discover a wide variety of scientific laws, such as Kepler's third law, Ohm's law, the conservation of momentum, and Joule's law. The heuristics BACON uses to discover the ideal gas law include noting constancies, finding linear relations, and defining theoretical terms. Other heuristics allow BACON to postulate intrinsic properties of objects and to reason by analogy. For example, if BACON finds a regularity in one set of parameters, it will attempt to generate the same regularity in a similar set of parameters. Since BACON's discovery procedure is state-space search, these heuristics allow it to reach solutions while visiting only a small portion of the search space. In the gas example, BACON comes up with the ideal gas law using a minimal number of experiments.

A better understanding of the science of scientific discovery may lead one day to programs that display true creativity. Much more work must be done in areas of science that BACON does not model, such as determining what data to gather, choosing (or creating) instruments to measure the data, and using analogies to previously understood phenomena. For a thorough discussion of scientific discovery programs, see Langley *et al.* [1987].

17.7.3 Clustering

A third type of discovery, called *clustering*, is very similar to induction, as we described it in Section 17.5. In inductive learning, a program learns to classify objects based on the labelings provided by a teacher. In clustering, no class labelings are provided. The program must discover for itself the natural classes that exist for the objects, in addition to a method for classifying instances.

AUTOCLASS [Cheeseman et al., 1988] is a program for automatically discovering classes in data. It hypothesizes a set of classes. For any given case, the program provides a set of probabilities that predict into which class(es) the case is likely to fall. In one application, AUTOCLASS found meaningful new classes of stars from their infrared spectral data. This was an instance of true discovery by computer, since the facts it discovered were previously unknown to astronomy. AUTOCLASS uses statistical Bayesian reasoning of the type discussed in Chapter 8.

17.8 ANALOGY

Analogy is a powerful inference tool. Our language and reasoning are laden with analogies. Consider the following sentences:

- Last month, the stock market was a roller coaster.
- Bill is like a fire engine.
- Problems in electromagnetism are just like problems in fluid flow.

Underlying each of these examples is a complicated mapping between what appear to be dissimilar concepts. For example, to understand the first sentence above, it is necessary to do two things: (1) pick out one key property of a roller coaster, namely that it travels up and down rapidly and (2) realize that physical travel is itself an analogy for numerical fluctuations (in stock prices). This is no easy trick. The space of possible analogies is very large. We do not want to entertain possibilities such as "the stock market is like a roller coaster because it is made of metal."

Lakoff and Johnson [1980] make the case that everyday language is filled with such analogies and metaphors. An AI program that is unable to grasp analogy will be difficult to talk to and, consequently, difficult to teach. Thus, analogical reasoning is an important factor in learning by advice taking. It is also important to learning in problem-solving.

Humans often solve problems by making analogies to things they already understand how to do. This process is more complex than storing macro-operators (as discussed in Section 17.4.2) because the old problem might be quite different from the new problem on the surface. The difficulty comes in determining what things are similar and what things are not. Two methods of analogical problem solving that have been studied in AI are *transformational* and *derivational* analogy.

17.8.1 Transformational Analogy

Suppose you are asked to prove a theorem in plane geometry. You might look for a previous theorem that is very similar and "copy" its proof, making substitutions when necessary. The idea is to transform a solution to a previous problem into a solution for the current problem. Figure 17.19 shows this process.

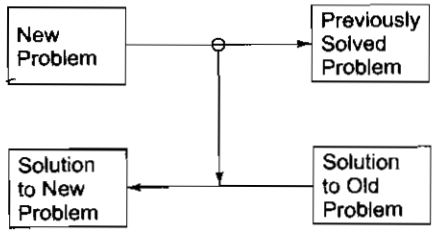
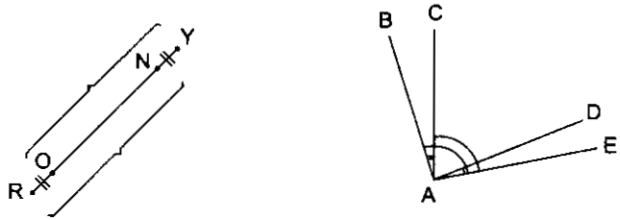


Fig. 17.19 *Transformational Analogy*

An example of transformational analogy is shown in Fig. 17.20 [Anderson and Kline, 1979]. The program has seen proofs about points and line segments; for example, it knows a proof that the line segment RN is exactly as long as the line segment OY, given that RO is exactly as long as NY. The program is now asked to prove a theorem about angles, namely that the angle BD is equivalent to the angle CE, given that angles BC and DE are equivalent. The proof about line segments is retrieved and transformed into a proof about angles by substituting the notion of line for point, angle for line segment, AB for R, AC for O, AD for N, and AE for Y.

Carbonell [1983] describes one method for transforming old solutions into new solutions. Whole solutions are viewed as states in a problem space called *T-space*. *T-operators* prescribe the methods of transforming

solutions (states) into other solutions. Reasoning by analogy becomes search in T-space: starting with an old solution, we use means-ends analysis or some other method to find a solution to the current problem.



Old Proof:		New Proof:
RO = NY	(given)	BAC = DAE
ON = ON	(reflexive)	CAD = CAD
RO + ON = ON + NY	(additive)	BAC + CAD = CAD + DAE
RN = OY	(transitive)	BAD = CAE

Fig. 17.20 *Solving a Problem by Transformational Analogy*

17.8.2 Derivational Analogy

Notice that transformational analogy does not look at *how* the old problem was solved: it only looks at the final solution. Often the twists and turns involved in solving an old problem are relevant to solving a new problem. The detailed history of a problem-solving episode is called its *derivation*. Analogical reasoning that takes these histories into account is called derivational analogy (see Fig. 17.21).

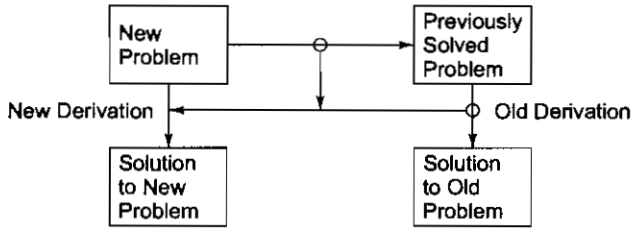


Fig. 17.21 *Derivational Analogy*

Carbonell [1986] claims that derivational analogy is a necessary component in the transfer of skills in complex domains. For example, suppose you have coded an efficient sorting routine in Pascal, and then you are asked to recode the routine in LISP. A line-by-line translation is not appropriate, but you will reuse the major structural and control decisions you made when you constructed the Pascal program. One way to model this behavior is to have a problem-solver "replay" the previous derivation and modify it when necessary. If the original reasons and assumptions for a step's existence still hold in the new problem, the step is copied over. If some assumption is no longer valid, another assumption must be found. If one cannot be found, then we can try to find justification for some alternative stored in the derivation of the original problem. Or perhaps we can try some step marked as leading to search failure in the original derivation, if the reasons to failure conditions are not valid in the current derivation.

Analogy in problem solving is a very open area of research. For a survey of recent see Hall [1989].

17.9 FORMAL LEARNING THEORY

Like many other AI problems, learning has attracted the attention of mathematicians and theoretical computer scientists. Inductive learning in particular has received con-siderable attention. Valiant [1984] describes a "theory of the learnable" which classifies problems by how difficult they are to learn. Formally, a device learns a concept if it can, given positive and negative examples, produce an algorithm that will classify future

examples to be [illegible]. The complexity of learning is a function of the error tolerance (h), the number of binary features present in the examples (t), and the size of the rule necessary to make the discrimination (f). If the number of training examples required is polynomial in h, t, and f, then the concept is said to be *learnable*.

Some interesting results have been demonstrated for concept learning. Consider the problem of learning conjunctive feature descriptions. For example, from the list of positive and negative examples of elephants shown in Fig. 17.22, we want to induce the description "gray, mammal, large." It has been shown that in conjunctive learning the number of randomly chosen training examples is proportional to the logarithm of the total number of features [Haussler, 1988; Littlestone, 1988].[6] Since very few training examples are needed to solve this induction problem, it is called *learnable*. Even if we restrict the learner *to positive* examples only, conjunctive learning can be achieved when the number of examples is linearly proportional to the number of attributes [Ehrenfeucht *et al.*, 1989]. Learning from positive examples only is a phenomenon not modeled by least-commitment inductive techniques such as version spaces. The introduction of the error tolerance h makes this possible: After all, even if all the elephants in our training set are gray, we may later encounter a genuine elephant that happens to be white. Fortunately, we can extend the size of our randomly sampled training set to ensure that the probability of misclassifying an elephant as something else (such as a polar bear) is an arbitrarily small $1/h$.

gray?	mammal?	large?	vegetarian?	wild?		
+	+	+	+	+	+	*(Elephant)*
+	+	+	−	+	+	*(Elephant)*
+	+	−	+	+	−	*(Mouse)*
−	+	+	+	+	−	*(Giraffe)*
+	−	+	−	+	−	*(Dinosaur)*
+	+	+	+	−	+	*(Elephant)*

Fig. 17.22 *Six Positive and Negative Examples of the Concept Elephant*

Formal techniques have been applied to a number of other learning problems. For example, given positive and negative examples of strings in some regular language, can we efficiently induce the finite automaton that produces all and only the strings in that language? The answer is no; an exponential number of computational steps is required [Kearns and Valiant, 1989].[7] However, if we allow the learner to make specific queries (e.g., "Is string x in the language?"), then the problem *is* learnable [Angluin, 1987].

It is difficult to tell how such mathematical studies of learning will affect the ways in which we solve AI problems in practice. After all, people are able to solve many exponentially hard problems by using knowledge to constrain the space of possible solutions. Perhaps mathematical theory will one day be used to quantify the use of such knowledge, but this prospect seems far off. For a critique of formal learning theory as well as some of the inductive techniques described in Section 17.5, see Amsterdam [1988].

17.10 NEURAL NET LEARNING AND GENETIC LEARNING

The very first efforts in machine learning tried to mimic animal learning at a neural level. These efforts were quite different from the symbolic manipulation methods we have seen so far in this chapter. Collections of idealized neurons were presented with stimuli and prodded into changing their behavior via forms of reward

[6] However, the number of examples must be *linear* in the number of *relevant* attributes, i.e., the number of attributes that appear in the learned conjunction.

[7] The proof of this result rests on some unproven hypotheses about the complexity of certain number theoretic functions.

and punishment. Researchers hoped that by imitating the learning mechanisms of animals, they might build learning machines from very simple parts. Such hopes proved elusive. However, the field of neural network learning has seen a resurgence in recent years, partly as a result of the discovery of powerful new learning algorithms. Chapter 18 describes these algorithms in detail.

While neural network models are based on a computational "brain metaphor," a number of other learning techniques make use of a metaphor based on evolution. In this work, learning occurs through a selection process that begins with a large population of random programs. Learning algorithms inspired by evolution are called" *genetic algorithms* [Holland, 1975; de Jong, 1988; Goldberg, 1989]. GAs have been dealt with in greater detail in Chapter 23.

SUMMARY

The most important thing to conclude from our study of automated learning is that learning itself is a problem-solving process. We can cast various learning strategies in terms of the methods of Chapters 2 and 3.

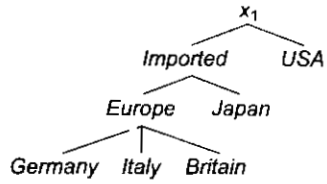
- Learning by taking advice
  - Initial state: high-level advice
  - Final state: an operational rule
  - Operators: unfolding definitions, case analysis, matching, etc.
- Learning from examples
  - Initial state: collection of positive and negative examples
  - Final state: concept description
  - Search algorithms: candidate elimination, induction of decision trees
- Learning in problem solving
  - Initial state: solution traces to example problems
  - Final state: new heuristics for solving new problems efficiently
  - Heuristics for search: generalization, explanation-based learning, utility considerations
- Discovery
  - Initial state: some environment
  - Final state: unknown
  - Heuristics for search: interestingness, analogy, etc.

A learning machine is the dream system of AI. As we have seen in previous chapters, the key to intelligent behavior is having a lot of knowledge. Getting all of that knowledge into a computer is a staggering task. One hope of sidestepping the task is to let computers acquire knowledge independently, as people do. We do not yet have programs that can extend themselves indefinitely. But we have discovered some of the reasons for our failure to create such systems. If we look at actual learning programs, we find that the more knowledge a program starts with, the more it can learn. Tbis finding is satisfying, in the sense that it corroborates our other discoveries about the power of knowledge. But it is also unpleasant, because it seems that fully self-extending systems are, for the present, still out of reach.

Research in machine learning has gone through several cycles of popularity. Timing is always an important consideration. A learning program needs to acquire new knowledge and new problem-solving abilities, but knowledge and problem-solving are topics still under intensive study. If we do not understand the nature of the thing we want to learn, learning is difficult. Not surprisingly, the most successful learning programs operate in fairly well-understood areas (like planning), and not in less well-understood areas (like natural language understanding).

EXERCISES

1. Would it be reasonable to apply Samuel's rote-learning procedure to chess? Why (not)?
2. Implement the candidate elimination algorithm for version spaces. Choose a concept space with several features (for example, the space of books, computers, animals, etc.) Pick a concept and demonstrate learning by presenting positive and negative examples of the concept.
3. In Section 17.5.2, the concept "Japanese economy car" was learned through the presentation of five positive and negative examples. Give a sequenee of *four* examples that accomplishes the same goal. In general, what properties of a positive example make it most useful? What makes a negative example most useful?
4. Recall the problem of learning disjunctive concepts in version spaces. We discussed learning a concept like "European car," where a European car was defined as a car whose *origin* was either *Germany, Italy,* or *Britain.* Suppose we expand the number of discrete values the slot *origin* might take to include the values *Europe* and *Imported.* Suppose further that we have the following *isa* hierarchy at our disposal:



   The diagram reflects facts such as "Japanese cars are a subset of imported cars" and "Italian cars are a subset of European cars." How could we modify the candidate elimination algorithm to take advantage of this knowledge? Propose new methods of updating tne sets *G* and *S* that would allow us to learn the concept "European car" in one pass through a set of adequate training examples.
5. AM exploited a set of 250 heuristics designed to guide AM's behavior toward interesting mathematical concepts. A classic work by Polya [1957] describes a set of heuristics for solving mathematical problems. Unfortunately, Polya's heuristics are not specified in enough detail to make them implementable in a program. In particular they lack precise descriptions of the situations in which they are appropriate (i.e., the left sides if they are viewed as productions). Examine some of Polya's rules and refine them so that they could be implemented in a problem-solving program with a structure similar to AM's.
6. Consider the problem of building a program to learn a grammar for a language such as English. Assume that such a program would be providad, as input, with a set of pairs, each consisting of a sentence and a representation of the meaning of the sentence. This is analogous to the experience of a child who hears a sentence and sees something at the same time. How could such a program be built using the techniques discussed in this chapter?

CHAPTER

20

EXPERT SYSTEMS

An expert is a man who has made all the mistakes which can be made in a very narrow field.

—**Niels Bohr**
(1885–1962), Danish physicist

Expert systems solve problems (such as the ones in Fig. 1.1) that are normally solved by human "experts." To solve expert-level problems, expert systems need access to a substantial domain knowledge base, which must be built as efficiently as possible. They also need to exploit one or more reasoning mechanisms to apply their knowledge to the problems they are given. Then they need a mechanism for explaining what they have done to the users who rely on them. One way to look at expert systems is that they represent applied AI in a very broad sense. They tend to lag several years behind research advances, but because they are tackling harder and harder problems, they will eventually be able to make use of all of the kinds of results that we have described throughout this book. So this chapter is in some ways a review of much of what we have already discussed.

The problems that expert systems deal with are highly diverse. There are some general issues that arise across these varying domains. But it also turns out that there are powerful techniques that can be defined for specific classes of problems. Recall that in Section 2.3.8 we introduced the notion of problem classification and we described some classes into which problems can be organized. Throughout this chapter we have occasion to return to this idea, and we see how some key problem characteristics play an important role in guiding the design of problem-solving systems. For example, it is now clear that tools that are developed to support one classification or diagnosis task are often useful for another, while different tools are useful for solving various kinds of design tasks.

20.1 REPRESENTING AND USING DOMAIN KNOWLEDGE

Expert systems are complex AI programs. Almost all the techniques that we described in Parts I and II have been exploited in at least one expert system. However, the most widely used way of representing domain knowledge in expert systems is as a set of production rules, which are often coupled with a frame system that defines the objects that occur in the rules. In Section 8.2, we saw one example of an expert system rule, which was taken from the MYCIN system. Let's look at a few additional examples drawn from some other

representative expert systems. All the rules we show are English versions of the actual rules that the systems use. Differences among these rules illustrate some of the important differences in the ways that expert systems operate.

Rl [McDermott, 1982; McDermott, 19841 (sometimes also called XCON) is a program that configures DEC VAX systems. Its rules look like this:

```
If: the most current active context is distributing
        massbus devices, and
    there is a single-port disk drive that has not been
        assigned to a massbus, and
    there are no unassigned dual-port disk drives, and
        the number of devices that each massbus should
        support is known, and
    there is a massbus that has been assigned at least
        one disk drive and that should support additional
        disk drives,
    and the type of cable needed to connect the disk drive
        to the previous device on the massbus is known
then: assign the disk drive to the massbus.
```

Notice that Rl's rules, unlike MYCIN's, contain no numeric measures of certainty. In the task domain with which Rl deals, it is possible to state exactly the correct thing to be done in each particular set of circumstances (although it may require a relatively complex set of antecedents to do so). One reason for this is that there exists a good deal of human expertise in this area. Another is that since Rl is doing a design task (in contrast to the diagnosis task performed by MYCIN), it is not necessary to consider all possible alternatives; one good one is enough. As a result, probabilistic information is not necessary in R1.

PROSPECTOR [Duda *et al.*. 1979; Hart *et al.*, 1978] is a program that provides advice on mineral exploration. Its rules look like this:

```
If: magnetite or pyrite in disseminated or veinlet form is present
then: (2, -4) there is favorable mineralization and texture
    for the propylitic stage.
```

In PROSPECTOR, each rule contains two confidence estimates. The first indicates the extent to which the presence of the evidence described in the condition part of the rule suggests the validity of the rule's conclusion. In the PROSPECTOR rule shown above, the number 2 indicates that the presence of the evidence is mildly encouraging. The second confidence estimate measures the extent to which the evidence is necessary to the validity of the conclusion, or stated another way, the extent to which the lack of the evidence indicates that the conclusion is not valid. In the example rule shown above, the number –4 indicates that the absence of the evidence is strongly discouraging for the conclusion.

DESIGN ADVISOR [Steele *et al.*, 1989] is a system that critiques chip designs. Its rules look like:

```
If: the sequential level count of ELEMENT is greater than 2,
    UNLESS the signal of ELEMENT is resetable
then: critique for poor resetability
DEFEAT: poor resetability of ELEMENT
due to: sequential level count of ELEMENT greater than 2
by: ELEMENT is directly resetable
```

https://hemanthrajhemu.github.io

The DESIGN ADVISOR gives advice to a chip designer, who can accept or reject the advice. If the advice is rejected, the system can exploit a justification-based truth maintenance system to revise its model of the circuit. The first rule shown here says that an element should be criticized for poor resetability if its sequential level count is greater than two, unless its signal is currently believed to be resetable. Resetability is a fairly common condition, so it is mentioned explicitly in this first rule. But there is also a much less common condition, called direct resetability. The DESIGN ADVISOR does not even bother to consider that condition unless it gets in trouble with its advice. At that point, it can exploit the second of the rules shown above. Specifically, if the chip designer rejects a critique about resetability and if that critique was based on a high level count, then the system will attempt to discover (possibly by asking the designer) whether the element is directly resetable. If it is, then the original rule is defeated and the conclusion withdrawn.

Reasoning with the Knowledge

As these example rules have shown, expert systems exploit many of the representation and reasoning mechanisms that we have discussed. Because these programs are usually written primarily as rule-based systems, forward chaining, backward chaining, or some combination of the two, is usually used. For example, MYCIN used backward chaining to discover what organisms were present; then it used forward chaining to reason from the organisms to a treatment regime. R1, on the other hand, used forward chaining. As the field of expert systems matures, more systems that exploit other kinds of reasoning mechanisms are being developed. The DESIGN ADVISOR is an example of such a system; in addition to exploiting rules, it makes extensive use of a justification-based truth maintenance system.

20.2 EXPERT SYSTEM SHELLS

Initially, each expert system that was built was created from scratch, usually in LISP. But, after several systems had been built this way, it became clear that these systems often had a lot in common. In particular, since the systems were constructed as a set of declarative representations (mostly rules) combined with an interpreter for those representations, it was possible to separate the interpreter from the domain-specific knowledge and thus to create a system that could be used to construct new expert systems by adding new knowledge corresponding to the new problem domain. The resulting interpreters are called *shells*. One influential example of such a shell is EMYCIN (for Empty MYCIN) [Buchanan and Shortliffe, 1984], which was derived from MYCIN.

There are now several commercially available shells that serve as the basis for many of the expert systems currently being built. These shells provide much greater flexibility in representing knowledge and in reasoning with it than MYCIN did. They typically support rules, frames, truth maintenance systems, and a variety of other reasoning mechanisms.

Early expert system shells provided mechanisms for knowledge representation, reasoning, and explanation. Later, tools for knowledge acquisition were added, as we see in Section 20.4. But as experience with using these systems to solve real world problems grew, it became clear that expert system shells needed to do something else as well. They needed to make it easy to integrate expert systems with other kinds of programs. Expert systems cannot operate in a vacuum, any more than their human counterparts can. They need access to corporate databases, and access to them needs to be controlled just as it does for other systems. They are often embedded within larger application programs that use primarily conventional programming techniques. So one of the important features that a shell must provide is an easy-to-use interface between an expert system that is written with the shell and a larger, probably more conventional, programming environment.

20.2 EXPLANATION

In order for an expert system to be an effective tool, people must be able to interact with it easily. To facilitate this interaction, the expert system must have the following two capabilities in addition to the ability to perform its underlying task:

- Explain its reasoning. In many of the domains in which expert systems operate, people will not accept results unless they have been convinced of the accuracy of the reasoning process that produced those results. This is particularly true, for example, in medicine, where a doctor must accept ultimate responsibility for a diagnosis, even if that diagnosis was arrived at with considerable help from a program. Thus it is important that the reasoning process used in such programs proceed in understandable steps and that enough meta-knowledge (knowledge about the reasoning process) be available so the explanations of those steps can be generated.
- Acquire new knowledge and modifications of old knowledge. Since expert systems derive their power from the richness of the knowledge bases they exploit, it is extremely important that those knowledge bases be as complete and as accurate as possible. But often there exists no standard codification of that knowledge; rather it exists only inside the heads of human experts. One way to get this knowledge into a program is through interaction with the human expert. Another way is to have the program learn expert behavior from raw data.

TEIRESIAS [Davis, 1982; Davis, 1977] was the first program to support explanation and knowledge acquisition. TEIRESIAS served as a front-end for the MYCIN expert system. A fragment of a TEIRESIAS-MYCIN conversation with a user (a doctor) is shown in Fig. 20.1. The program has asked for a piece of information that it needs in order to continue its reasoning. The doctor wants to know why the program wants the information, and later asks how the program arrived at a conclusion that it claimed it had reached.

An important premise underlying TEIRESIAS's approach to explanation is that the behavior of a program can be explained simply by referring to a trace of the program's execution. There are ways in which this assumption limits the kinds of explanations that can be produced, but it does minimize the overhead involved in generating each explanation. To understand how TEIRESIAS generates explanations of MYCIN's behavior, we need to know how that behavior is structured.

MYCIN attempts to solve its goal of recommending a therapy for a particular patient by first finding the cause of the patient's illness. It uses its production rules to reason backward from goals to clinical observations. To solve the top-level diagnostic goal, it looks for rules whose right sides suggest diseases. It then uses the left sides of those rules (the preconditions) to set up subgoals whose success would enable the rules to be invoked. These subgoals are again matched against rules, and their preconditions are used to set up additional subgoals. Whenever a precondition describes a specific piece of clinical evidence, MYCIN uses that evidence if it already has access to it. Otherwise, it asks the user to provide the information. In order that MYCIN's requests for information will appear coherent to the user, the actual goals that MYCIN sets up are often more general than they need be to satisfy the preconditions of an individual rule. For example, if a precondition specifies that the identity of an organism is *X*, MYCIN will set up the goal "infer identity." This approach also means that if another rule mentions the organism's identity, no further work will be required, since the identity will be known.

We can now return to the trace of TEIRESIAS-MYCIN's behavior shown in Fig. 20.1. The first question that the user asks is a "WHY" question, which is assumed to mean "Why do you need to know that?" Particularly for clinical tests that are either expensive or dangerous, it is important for the doctor to be convinced that the information is really needed before ordering the test. (Requests for sensitive or confidential information present similar difficulties.) Because MYCIN is reasoning backward, the question can easily be answered by examining the goal tree. Doing so provides two kinds of information:

- What higher-level question might the system be able to answer if it had the requested piece of information? (In this case, it could help determine the category of ORGANISM-1.)
- What other information does the system already have that makes it think that the requested piece of knowledge would help? (In this case, facts [2.1] to [2.4].)

```
Is the patient's illness with ORGANISM-1 a hospital-acquired infection?
** WHY
[i.e., WHY is it important to determine whether or not the
infection with ORGANISM-1 was acquired while the patient
was hospitalized?]
[2.0]  This will aid in determining the category of
       ORGANISM-1. It has already been established that
[2.1]  the gram stain of ORGANISM-1 is gramneg, and
[2.2]  the morphology of ORGANISM-1 is rod, and
[2.3]  the aerobicity of ORGANISM-1 is facultative
       Therefore, if
[2.4]  the infection with ORGANISM-1 was not acquired
       while the patient was hospitalized, then
there is strongly suggestive evidence (.8) that the
category of ORGANISM-1 is enterobacteriaceae
 [ALSO: there is weakly suggestive evidence (.1) that
 the identity of ORGANISM-1 is pseudomonas-aeruginosa]
 [RULE037]
** HOW [2.3]
[i.e., HOW was it established that the aerobicity of
 ORGANISM-1 is facultative?]
The following were used:
[3.1]  RULE027 indicated there is strongly suggestive
       evidence (.8) that the aerobicity of ORGANISM-1
       is facultative.
[3.2]  RULE033 indicated there is weakly suggestive
       evidence (.2) that the aerobicity of ORGANISM-1
       is not facultative.
Since this gave a cumulative CF of (.6) for facultative,
the evidence indicates that the aerobicity of ORGANISM-1
is facultative.
```

Fig. 20.1 *A Portion of a Dialogue with TEIRESIAS*

When TEIRESIAS provides the answer to the first of these questions, the user may be satisfied or may want to follow the reasoning process back even further. The user can do that by asking additional "WHY" questions.

When TEIRESIAS provides the answer to the second of these questions and tells the user what it already believes, the user may want to know the basis for those beliefs. The user can ask this with a "HOW" question, which TEIRESIAS will interpret as "How did you know that?" This question also can be answered by looking at the goal tree and chaining backward from the stated fact to the evidence that allowed a rule that determined the fact to fire. Thus we see that by reasoning backward from its top-level goal and by keeping track of the entire tree that it traverses in the process, TEIRESIAS- MYCIN can do a fairly good job of justifying its reasoning to a human user. For more details of this process, as well as a discussion of some of its limitations, see Davis [1982].

The production system model is very general, and without some restrictions, it is hard to support all the kinds of explanations that a human might want. If we focus on a particular type of problem solving, we can ask more probing questions. For example, SALT [Marcus and McDermott, 1989] is a knowledge acquisition program used to build expert systems that design artifacts through a *propose-and-revise* strategy. SALT is

capable of answering questions like WHY-NOT ("why didn't you assign value x to this parameter?") and WHAT-IF ("what would happen if you did?"). A human might ask these questions in order to locate incorrect or missing knowledge in the system as a precursor to correcting it. We now turn to ways in which a program such as SALT can support the process of building and refining knowledge.

20.4 KNOWLEDGE ACQUISITION

How are expert systems built? Typically, a knowledge engineer interviews a domain expert to elucidate expert knowledge, which is then translated into rules. After the initial system is built, it must be iteratively refined until it approximates expert-level performance. This process is expensive and time-consuming, so it is worthwhile to look for more automatic ways of constructing expert knowledge bases. While no totally automatic knowledge acquisition systems yet exist, there are many programs that interact with domain experts to extract expert knowledge efficiently. These programs provide support for the following activities:

- Entering knowledge
- Maintaining knowledge base consistency
- Ensuring knowledge base completeness

The most useful knowledge acquisition programs are those that are restricted to a particular problem-solving paradigm, e.g., diagnosis or design. It is important to be able to enumerate the roles that knowledge can play in the problem-solving process. For example, if the paradigm is diagnosis, then the program can structure its knowledge base around symptoms, hypotheses, and causes. It can identify symptoms for which the expert has not yet provided causes. Since one symptom may have multiple causes, the program can ask for knowledge about how to decide when one hypothesis is better than another. If we move to another type of problem-solving, say designing artifacts, then these acquisition strategies no longer apply, and we must look for other ways of profitably interacting with an expert. We now examine two knowledge acquisition systems in detail.

MOLE [Eshelman, 1988] is a knowledge acquisition system for heuristic classification problems, such as diagnosing diseases. In particular, it is used in conjunction with the *cover-and-differentiate* problem-solving method. An expert system produced by MOLE accepts input data, comes up with a set of candidate explanations or classifications that cover (or explain) the data, then uses differentiating knowledge to determine which one is best. The process is iterative, since explanations must themselves be justified, until ultimate causes are ascertained.

MOLE interacts with a domain expert to produce a knowledge base that a system called MOLE-p (for MOLE-performance) uses to solve problems. The aequisition proceeds through several steps:

1. Initial knowledge base construction. MOLE asks the expert to list common symptoms or complaints that might require diagnosis. For each symptom, MOLE prompts for a list of possible explanations. MOLE then iteratively seeks out higher-level explanations until it comes up with a set of ultimate causes. During this process, MOLE builds an influence network similar to the belief networks we saw in Chapter 8.

   Whenever an event has multiple explanations, MOLE tries to determine the conditions under which one explanation is correct. The expert provides *covering* knowledge, that is, the knowledge that a hypothesized event might be the cause of a certain symptom. MOLE then tries to infer *anticipatory* knowledge, which says that if the hypothesized event does occur, then the symptom will definitely appear. This knowledge allows the system to rule out certain hypotheses on the basis that specific symptoms are absent.
2. Refinement of the knowledge base. MOLE now tries to identify the weaknesses of the knowledge base. One approach is to find holes and prompt the expert to fill them. It is difficult, in general, to know whether a knowledge base is complete, so instead MOLE lets the expert watch MOLE-p solving sample

problems. Whenever MOLE-p makes an incorrect diagnosis, the expert adds new knowledge. There are several ways in which MOLE-p can reach the wrong conclusion. It may incorrectly reject a hypothesis because it does not feel that the hypothesis is needed to explain any symptom. It may advance a hypothesis because it is needed to explain some otherwise inexplicable hypothesis. Or it may lack differentiating knowledge for choosing between alternative hypotheses.

For example, suppose we have a patient with symptoms A and B. Further suppose that symptom A could be caused by events X and Y, and that symptom B can be caused by Y and Z. MOLE-p might conclude Y, since it explains both A and B. If the expert indicates that this decision was incorrect, then MOLE will ask what evidence should be used to prefer X and/or Z over Y.

MOLE has been used to build systems that diagnose problems with car engines, problems in steel-rolling mills, and inefficiencies in coal-burning power plants. For MOLE to be applicable, however, it must be possible to preenumerate solutions or classifications. It must also be practical to encode the knowledge in terms of covering and differentiating.

But suppose our task is to design an artifact, for example, an elevator system. It is no longer possible to preenumerate all solutions. Instead, we must assign values to a large number of parameters, such as the width of the platform, the type of door, the cable weight, and the cable strength. These parameters must be consistent with each other, and they must result in a design that satisfies external constraints imposed by cost factors, the type of building involved, and expected payloads.

One problem-solving method useful for design tasks is called *propose-and-revise*. Propose-and-revise systems build up solutions incrementally. First, the system proposes an extension to the current design. Then it checks whether the extension violates any global or local constraints. Constraint violations are then fixed, and the process repeats. It turns out that domain experts are good at listing overall design constraints and at providing local constraints on individual parameters, but not so good at explaining how to arrive at global solutions. The SALT program [Marcus and McDermott, 1989] provides mechanisms for elucidating this knowledge from the expert.

Like MOLE, SALT builds a dependency network as it converses with the expert. Each node stands for a value of a parameter that must be acquired or generated. There are three kinds of links: *contributes-to, constrains,* and *suggests-revision-of.* Associated with the first type of link are procedures that allow SALT to generate a value for one parameter based on the value of another. The second type of link, *constrains,* rules out certain parameter values. The third link, *suggests-revision-of,* points to ways in which a constraint violation can be fixed. SALT uses the following heuristics to guide the acquisition process:

1. Every noninput node in the network needs at least one *contributes-to* link coming into it. If links are missing, the expert is prompted to fill them in.
2. No *contributes-to* loops are allowed in the network. Without a value for at least one parameter in the loop, it is impossible to compute values for any parameter in that loop. If a loop exists, SALT tries to transform one of the *contributes-to* links into a *constrains* link.
3. Constraining links should have *suggests-revision-of* links associated with them. These include *constrains* links that are created when dependency loops are broken.

Control knowledge is also important. It is critical that the system propose extensions and revisions that lead toward a design solution. SALT allows the expert to rate revisions in terms of how much trouble they tend to produce.

SALT compiles its dependency network into a set of production rules. As with MOLE, an expert can watch the production system solve problems and can override the system's decision. At that point, the knowledge base can be changed or the override can be logged for future inspection.

The process of interviewing a human expert to extract expertise presents a number of difficulties, regardless of whether the interview is conducted by a human or by a machine. Experts are surprisingly inarticulate when it comes to how they solve problems. They do not seem to have access to the low-level details of what they do and are especially inadequate suppliers of any type of statistical information. There is, therefore, a great deal of interest in building systems that automatically induce their own rules by looking at sample problems and solutions. With inductive techniques, an expert needs only to provide the conceptual framework for a problem and a set of useful examples.

For example, consider a bank's problem in deciding whether to approve a loan. One approach to automating this task is to interview loan officers in an attempt to extract their domain knowledge. Another approach is to inspect the record of loans the bank has made in the past and then try to generate automatically rules that will maximize the number of good loans and minimize the number of bad ones in the future.

META-DENDRAL [Mitchell, 1978] was the first program to use learning techniques to construct rules for an expert system automatically. It built rules to be used by DENDRAL, whose job was to determine the structure of complex chemical compounds. META-DENDRAL was able to induce its rules based on a set of mass spectrometry data; it was then able to identify molecular structures with very high accuracy. META-DENDRAL used the version space learning algorithm, which we discussed in Chapter 17. Another popular method for automatically constructing expert systems is the induction of decision trees, data structures we described in Section 17.5.3. Decision tree expert systems have been built for assessing consumer credit applications, analyzing hypothyroid conditions, and diagnosing soybean diseases, among many other applications.

Statistical techniques, such as multivariate analysis, provide an alternative approach to building expert-level systems. Unfortunately, statistical methods do not produce concise rules that humans can understand. Therefore it is difficult for them to explain their decisions.

For highly structured problems that require deep causal chains of reasoning, learning techniques are presently inadequate. There is, however, a great deal of research activity in this area, as we saw in Chapter 17.

SUMMARY

Since the mid-1960s, when work began on the earliest of what are now called expert systems, much progress has been made in the construction of such programs. Experience gained in these efforts suggests the following conclusions:

- These systems derive their power from a great deal of domain-specific knowledge, rather than from a single powerful technique.
- In successful systems, the required knowledge is about a particular area and is well defined. This contrasts with the kind of broad, hard-to-define knowledge that we call common sense. It is easier to build expert systems than ones with common sense.
- An expert system is usually built with the aid of one or more experts, who must be willing to spend a great deal of effort transferring their expertise to the system.
- Transfer of knowledge takes place gradually through many interactions between the expert and the system. The expert will never get the knowledge right or complete the first time.
- The amount of knowledge that is required depends on the task. It may range from forty rules to thousands.
- The choice of control structure for a particular system depends on specific characteristics of the system.
- It is possible to extract the nondomain-specific parts from existing expert systems and use them as tools for building new systems in new domains.

Four major problems facing current expert systems are:

- Brittleness—Because expert systems only have access to highly specific domain knowledge, they cannot fall back on more general knowledge when the need arises. For example, suppose that we make a mistake in entering data for a medical expert system, and we describe a patient who is 130 years old and weighs 40 pounds. Most systems would not be able to guess that we may have reversed the two fields since the values aren't very plausible. The CYC system, which we discussed in Section 10.3, represents one attempt to remedy this problem by providing a substrate of common sense knowledge on which specific expert systems can be built.
- Lack of Meta-Knowledge—Expert systems do not have very sophisticated knowledge about their own operation. They typically cannot reason about their own scope and limitations, making it even more difficult to deal with the brittleness problem.
- Knowledge Acquisition—Despite the development of the tools that we described in Section 20.4, acquisition still remains a major bottleneck in applying expert systems technology to new domains.
- Validation—Measuring the performance of an expert system is difficult because we do not know how to quantify the use of knowledge. Certainly it is impossible to present formal proofs of correctness for expert systems. One thing we can do is pit these systems against human experts on real-world problems. For example, MYCIN participated in a panel of experts in evaluating ten selected meningitis cases, scoring higher than any of its human competitors [Buchanan, 1982]

There are many issues in the design and implementation of expert systems that we have not covered. For example, there has been a substantial amount of work done in the area of real-time expert systems [Laffey *et al.*, 1988]. For more information on the whole area of expert systems and to get a better feel for the kinds of applications that exist, look at Weiss and Kulikowski [1984], Harmon and King [1985]. Rauch-Hindin [1986], Waterman [1986], and Prerau [1990].

EXERCISES

1. Rule-based systems often contain rules with several conditions in their left sides.
   (a) Why is this true in MYCIN?
   (b) Why is this true in R1?
2. Contrast expert systems and neural networks (Chapter 18) in terms of knowledge representation, knowledge acquisition, and explanation. Give one domain in which the expert system approach would be more promising and one domain in which the neural network approach would be more promising.