CHAPTER

6

REPRESENTING KNOWLEDGE USING RULES

To be useful, a system has to do more than just correctly perform some task.

—John McDermott,
AI Researcher

In this chapter, we discuss the use of rules to encode knowledge. This is a particularly important issue since rule-based reasoning systems have played a very important role in the evolution of AI from a purely laboratory science into a commercially significant one, as we see later in Chapter 20.

We have already talked about rules as the basis for a search program. But we gave little consideration to the way knowledge about the world was represented in the rules (although we can see a simple example of this in Section 4.2). In particular, we have been assuming that search control knowledge was maintained completely separately from the rules themselves. We will now relax that assumption and consider a set of rules to represent both knowledge about relationships in the world, as well as knowledge about how to solve problems using the content of the rules.

6.1 PROCEDURAL VERSUS DECLARATIVE KNOWLEDGE

Since our discussion of knowledge representation has concentrated so far on the use of logical assertions, we use logic as a starting point in our discussion of rule-based systems.

In the previous chapter, we viewed logical assertions as declarative representations of knowledge. A *declarative representation* is one in which knowledge is specified, but the use to which that knowledge is to be put is not given. To use a declarative representation, we must augment it with a program that specifies what is to be done to the knowledge and how. For example, a set of logical assertions can be combined with a resolution theorem prover to give a complete program for solving problems. There is a different way, though, in which logical assertions can be viewed, namely as a *program,* rather than as *data* to a program. In this view, the implication statements define the legitimate reasoning paths and the atomic assertions provide the starting points (or, if we reason backward, the ending points) of those paths. These reasoning paths define the possible execution paths of the program in much the same way that traditional control constructs, such as *if-then-else.* define the execution paths through traditional programs. In other words, we could view logical assertions as procedural representations of knowledge. A *procedural representation* is one in which the control information that is necessary to use the knowledge is considered to be embedded in the knowledge itself. To use a procedural representation, we need to augment it with an interpreter that follows the instructions given in the knowledge.

Actually, viewing logical assertions as code is not a very radical idea, given that all programs are really data to other programs that interpret (or compile) and execute them. The real difference between the declarative and the procedural views of knowledge lies in where control information resides. For example, consider the knowledge base:

$man(Marcus)$
$man(Caesar)$
$person(Cleopatra)$
$\forall x : man(x) \rightarrow person(x)$

Now consider trying to extract from this knowledge base the answer to the question

$\exists y : person(y)$

We want to bind y to a particular value for which *person* is true. Our knowledge base justifies any of the following answers:

$y = Marcus$
$y = Caesar$
$y = Cleopatra$

Because there is more than one value that satisfies the predicate, but only one value is needed, the answer to the question will depend on the order in which the assertions are examined during the search for a response. If we view the assertions as declarative, then they do not themselves say anything about how they will be examined. If we view them as procedural, then they do. Of course, nondeterministic programs are possible — for example, the concurrent and parallel programming constructs described in Dijkstra [1976], Hoare [1985], and Chandy and Misra [1989]. So, we could view these assertions as a nondeterministic program whose output is simply not defined. If we do this, then we have a "procedural" representation that actually contains no more information than does the "declarative" form. But most systems that view knowledge as procedural do not do this. The reason for this is that, at least if the procedure is to execute on any sequential or on most existing parallel machines, some decision must be made about the order in which the assertions will be examined. There is no hardware support for randomness. So if the interpreter must have a way of deciding, there is no real reason not to specify it as part of the definition of the language and thus to define the meaning of any particular program in the language. For example, we might specify that assertions will be examined in the order in which they appear in the program and that search will proceed depth-first, by which we mean that if a new subgoal is established then it will be pursued immediately and other paths will only be examined if the new one fails. If we do that, then the assertions we gave above describe a program that will answer our question with

$y = Cleopatra$

To see clearly the difference between declarative and procedural representations, consider the following assertions:

$man(Marcus)$
$man(Caesar)$
$\forall x : man(x) \rightarrow person(x)$
$person(Cleopatra)$

Viewed declaratively, this is the same knowledge base that we had before. All the same answers are supported by the system and no one of them is explicitly selected. But viewed procedurally, and using the control model we used to get *Cleopatra* as our answer before, this is a different knowledge base since now the answer to our question is *Marcus.* This happens because the first statement that can achieve the *person* goal is the inference rule $\forall x : man(x) \rightarrow person(x)$. This rule sets up a subgoal to find a man. Again the statements are examined from the beginning, and now *Marcus* is found to satisfy the subgoal and thus also the goal. So *Marcus* is reported as the answer.

It is important to keep in mind that although we have said that a procedural representation encodes control information in the knowledge base, it does so only to the extent that the interpreter for the knowledge base recognizes that control information. So we could have gotten a different answer to the *person* question by leaving our original knowledge base intact and changing the interpreter so that it examines statements from last to first (but still pursuing depth-first search). Following this control regime, we report *Caesar* as our answer.

There has been a great deal of controversy in AI over whether declarative or procedural knowledge representation frameworks are better. There is no clearcut answer to the question. As you can see from this discussion, the distinction between the two forms is often very fuzzy. Rather than try to answer the question of which approach is better, what we do in the rest of this chapter is to describe ways in which rule formalisms and interpreters can be combined to solve problems. We begin with a mechanism called *logic programming,* and then we consider more flexible structures for rule-based systems.

6.2 LOGIC PROGRAMMING

Logic programming is a programming language paradigm in which logical assertions are viewed as programs, as described in the previous section. There are several logic programming systems in use today, the most popular of which is PROLOG [Clocksin and Mellish, 1984; Bratko, 1986]. Programming in PROLOG has been described in more detail in Chapter 25. A PROLOG program is described as a series of logical assertions, each of which is a *Horn clause.*[1] A Horn clause is a clause (as defined in Section 5.4.1) that has at most one positive literal. Thus p, $\neg p \vee q$, and $p \rightarrow q$ are all Horn clauses. The last of these does not look like a clause

$\forall x : pet(x) \wedge small(x) \rightarrow apartmentpet(x)$
$\forall x : cat(x) \vee dog(x) \rightarrow pet(x)$
$\forall x : poodle(x) \rightarrow dog(x) \wedge small(x)$
$poodle(ftujfy)$

A Representation In Logic

```
apartmentpet(X) :- pet(X), small(X).
pet(X) :- cat(X).
pet(X) :- dog(X).
dog(X) :- poodle (X) .
small(X) :- poodle(X).
poodle(fluffy).
```

A Representation in PROLOG

Fig. 6.1 *A Declarative and a Procedural Representation*

[1] Programs written in pure PROLOG are composed only of Horn clauses. PROLOG, as an actual programming language, however, allows departures from Horn clauses. In the rest of this section, we limit our discussion to pure PROLOG.

it would return FALSE because it is unable to prove that Fluffy is a cat. Unfortunately, this program returns the same answer when given the goal even though the program knows nothing about Mittens and specifically knows nothing that might prevent Mittens from being a cat. Negation by failure requires that we make what is called the *closed world assumption,* which states that all relevant, true assertions are contained in our knowledge base or are derivable from assertions that are so contained. Any assertion that is not present can therefore be assumed to be false. This assumption, while often justified, can cause serious problems when knowledge bases are incomplete. We discuss this issue further in Chapter 7.

There is much to say on the topic of PROLOG-style versus LISP-style programming. A great advantage of logic programming is that the programmer need only specify rules and facts since a search engine is built directly into the language. The disadvantage is that the search control is fixed. Although it is possible to write PROLOG code that uses search strategies other than depth-first with backtracking, it is difficult to do so. It is even more difficult to apply domain knowledge to constrain a search. PROLOG does allow for rudimentary control of search through a non-logical operator called *cut.* A cut can be inserted into a rule to specify a point that may not be backtracked over.

More generally, the fact that PROLOG programs must be composed of a restricted set of logical operators can be viewed as a limitation of the expressiveness of the language. But the other side of the coin is that it is possible to build PROLOG compilers that produce very efficient code.

In the rest of this chapter, we retain the rule-based nature of PROLOG, but we relax a number of PROLOG'S design constraints, leading to more flexible rule-based architectures. Programming in PROLOG has been explained in more detail later in Chapter 25.

6.3 FORWARD VERSUS BACKWARD REASONING

The object of a search procedure is to discover a path through a problem space from an initial configuration to a goal state. While PROLOG only searches from a goal state, there are actually two directions in which such a search could proceed:

- Forward, from the start states
- Backward, from the goal states

The production system model of the search process provides an easy way of viewing forward and backward reasoning as symmetric processes. Consider the problem of solving a particular instance of the 8-puzzle. The rules to be used for solving the puzzle can be written as shown in Fig. 6.2. Using those rules we could attempt to solve the puzzle shown back in Fig. 2.12 in one of two ways:

Assume the areas of the tray are numbered:

1	2	3
4	5	6
7	8	9

Square 1 empty and Square 2 contains tile $n \rightarrow$
  Square 2 empty and Square 1 contains tile n
Square 1 empty and Square 4 contains tile $n \rightarrow$
  Square 4 empty and Square 1 contains tile n
Square 2 empty and Square 1 contains tile $n \rightarrow$
  Square 1 empty and Square 2 contains tile n

Fig. 6.2 *A Sample of the Rules for Solving the 8-Puzzle*

- *Reason forward from the initial states.* Begin building a tree of move sequences that might be solutions by starting with the initial configuration(s) at the root of the tree. Generate the next level of the tree by finding all the rules whose *left* sides match the root node and using their right sides to create the new configurations. Generate the next level by taking each node generated at the previous level and applying to it all of the rules whose left sides match it. Continue until a configuration that matches the goal state is generated.

- *Reason backward from the goal states.* Begin building a tree of move sequences that might be solutions by starting with the goal configuration(s) at the root of the tree. Generate the next level of the tree by finding all the rules whose *right* sides match the root node. These are all the rules that, if only we could apply them, would generate the state we want. Use the left sides of the rules to generate the nodes at this second level of the tree. Generate the next level of the tree by taking each node at the previous level and finding all the rules whose right sides match it. Then use the corresponding left sides to generate the new nodes. Continue until a node that matches the initial state is generated. This method of reasoning backward from the desired final state is often called *goal-directed reasoning.*

Notice that the same rules can be used both to reason forward from the initial state and to reason backward from the goal state. To reason forward, the left sides (the preconditions) are matched against the current state and the right sides (the results) are used to generate new nodes until the goal is reached. To reason backward, the right sides are matched against the current node and the left sides are used to generate new nodes representing new goal states to be achieved. This continues until one of these goal states is matched by an initial state.

In the case of the 8-puzzle, it does not make much difference whether we reason forward or backward; about the same number of paths will be explored in either case. But this is not always true. Depending on the topology of the problem space, it may be significantly more efficient to search in one direction rather than the other.

Four factors influence, the question of whether it is better to reason forward or backward:

- Are there more possible start states or goal states? We would like to move from the smaller set of states to the larger (and thus easier to find) set of states.
- In which direction is the branching factor (i.e., the average number of nodes that can be reached directly from a single node) greater? We would like to proceed in the direction with the lower branching factor.
- Will the program be asked to justify its reasoning process to a user? If so, it is important to proceed in the direction that corresponds more closely with the way the user will think.
- What kind of event is going to trigger a problem-solving episode? If it is the arrival of a new fact, forward reasoning makes sense. If it is a query to which a response is desired, backward reasoning is more natural.

A few examples make these issues clearer. It seems easier to drive from an unfamiliar place home than from home to an unfamiliar place. Why is this? The branching factor is roughly the same in both directions (unless one-way streets are laid out very strangely). But for the purpose of finding our way around, there are many more locations that count as being home than there are locations that count as the unfamiliar target place. Any place from which we know how to get home can be considered as equivalent to home. If we can get to any such place, we can get home easily, But in order to find a route from where we are to an unfamiliar place, we pretty much have to be already at the unfamiliar place. So in going toward the unfamiliar place, we are aiming at a much smaller target than in going home. This suggests that if our starting position is home and our goal position is the unfamiliar place, we should plan our route by reasoning backward from the unfamiliar place.

On the other hand, consider the problem of symbolic integration. The problem space is the set of formulas, some of which contain integral expressions. The start state is a particular formula containing some integral expression. The desired goal state is a formula that is equivalent to the initial one and that does not contain any integral expressions. So we begin with a single easily identified start state and a huge number of possible goal states. Thus to solve this problem, it is better to reason forward using the rules for integration to try to generate an integral-free expression than to start with arbitrary integral-free expressions, use the rules for differentiation, and try to generate the particular integral we are trying to solve. Again we want to head toward the largest target; this time that means chaining forward.

These two examples have illustrated the importance of the relative number of start states to goal states in determining the optimal direction in which to search when the branching factor is approximately the same in both directions. When the branching factor is not the same, however, it must also be taken into account.

Consider again the problem of proving theorems in some particular domain of mathematics. Our goal state is the particular theorem to be proved. Our initial states are normally a small set of axioms. Neither of these sets is significantly bigger than the other. But consider the branching factor in each of the two directions. From a small set of axioms we can derive a very large number of theorems. On the other hand, this large number of theorems must go back to the small set of axioms. So the branching factor is significantly greater going forward from the axioms to the theorems than it is going backward from theorems to axioms. This suggests that it would be much better to reason backward when trying to prove theorems. Mathematicians have long realized this [Polya, 1957], as have the designers of theorem-proving programs.

The third factor that determines the direction in which search should proceed is the need to generate coherent justifications of the reasoning process as it proceeds. This is often crucial for the acceptance of programs for the performance of very important tasks. For example, doctors are unwilling to accept the advice of a diagnostic program that cannot explain its reasoning to the doctors' satisfaction. This issue was of concern to the designers of MYCIN [Shortliffe, 1976], a program that diagnoses infectious diseases. It reasons backward from its goal of determining the cause of a patient's illness. To do that, it uses rules that tell it such things as "If the organism has the following set of characteristics as determined by the lab results, then it is likely that it is organism *x*." By reasoning backward using such rules, the program can answer questions like "Why should I perform that test you just asked for?" with such answers as "Because it would help to determine whether organism *x* is present." (For a discussion of the explanation capabilities of MYCIN, see Chapter 20.)

Most of the search techniques described in Chapter 3 can be used to search either forward or backward. By describing the search process as the application of a set of production rules, it is easy to describe the specific search algorithms without reference to the direction of the search.[2]

We can also search both forward from the start state and backward from the goal simultaneously until two paths meet somewhere in between. This strategy is called *bidirectional search.* It seems appealing if the number of nodes at each step grows exponentially with the number of steps that have been taken. Empirical results [Pohl, 1971] suggest that for blind search, this divide-and-conquer strategy is indeed effective. Unfortunately, other results [Pohl, 1971; de Champeaux and Sint, 1977] suggest that for informed, heuristic search it is much less likely to be *so.* Figure 6.3 shows why bidirectional search may be ineffective. The two searches may pass each other, resulting in more work than it would have taken for one of them, on its own, to have finished. However, if individual forward and backward steps are performed as specified by a program that has been carefully constructed to exploit each in exactly those situations where it can be the most profitable, the results can be more encouraging. In fact, many successful AI applications have been written using a combination of forward and backward reasoning, and most AI programming environments provide explicit support for such hybrid reasoning.

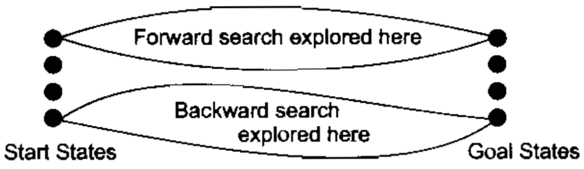


Fig. 6.3 *A Bad Use of Heuristic Bidirectional Search*

Although in principle the same set of rules can be used for both forward and backward reasoning, in practice it has proved useful to define two classes of rules, each of which encodes a particular kind of knowledge.

- Forward rules, which encode knowledge about how to respond to certain input configurations.
- Backward rules, which encode knowledge about how to achieve particular goals.

[2] One exception to this is the means-ends analysis technique, described in Section 3.6, which proceeds not by making successive steps in a single direction but by reducing differences between the current and the goal states, and, as a result, sometimes reasoning backward and sometimes forward.

By separating rules into these two classes, we essentially add to each rule an additional piece of information, namely, how it should be used in problem-solving. In the next three sections, we describe in more detail the two kinds of rule systems and how they can be combined.

6.3.1 Backward-Chaining Rule Systems

Backward-chaining rule systems, of which PROLOG is an example, are good for goal-directed problem-solving. For example, a query system would probably use backward chaining to reason about and answer user questions.

In PROLOG, rules are restricted to Horn clauses. This allows for rapid indexing because all of the rules for deducing a given fact share the same rule head. Rules are matched with the unification procedure. Unification tries to find a set of bindings for variables to equate a (sub)goal with the head of some rule. Rules in a PROLOG program are matched in the order in which they appear.

Other backward-chaining systems allow for more complex rules. In MYCIN, for example, rules can be augmented with probabilistic certainty factors to reflect the fact that some rules are more reliable than others. We discuss this in more detail in Chapter 8.

6.3.2 Forward-Chaining Rule Systems

Instead of being directed by goals, we sometimes want to be directed by incoming data. For example, suppose you sense searing heat near your hand. You are likely to jerk your hand away. While this could be construed as goal-directed behavior, it is modeled more naturally by the recognize-act cycle characteristic of forward-chaining rule systems. In forward-chaining systems, left sides of rules are matched against the state description. Rules that match dump their right-hand side assertions into the state, and the process repeats.

Matching is typically more complex for forward-chaining systems than backward ones. For example, consider a rule that checks for some condition in the state description and then adds an assertion. After the rule fires, its conditions are probably still valid, so it could fire again immediately. However, we will need some mechanism to prevent repeated firings, especially if the state remains unchanged.

While simple matching and control strategies are possible, most forward-chaining systems (e.g., OPS5 [Brownston *et al.*, 1985]) implement highly efficient matchers and supply several mechanisms for preferring one rule over another. We discuss matching in more detail in the next section.

6.3.3 Combining Forward and Backward Reasoning

Sometimes certain aspects of a problem are best handled via forward chaining and other aspects by backward chaining. Consider a forward-chaining medical diagnosis program. It might accept twenty or so facts about a patient's condition, then forward chain on those facts to try to deduce the nature and/or cause of the disease. Now suppose that at some point, the left side of a rule was *nearly* satisfied—say, nine out of ten of its preconditions were met. It might be efficient to apply backward reasoning to satisfy the tenth precondition in a directed manner, rather than wait for forward chaining to supply the fact by accident. Or perhaps the tenth condition requires further medical tests. In that case, backward chaining can be used to query the user.

Whether it is possible to use the same rules for both forward-and backward reasoning also depends on the form of the rules themselves. If both left sides and right sides contain pure assertions, then forward chaining can match assertions on the left side of a rule and add to the state description the assertions on the right side. But if arbitrary procedures are allowed as the right sides of rules, then the rules will not be reversible. Some production languages allow only reversible rules; others do not. When irreversible rules are used, then a commitment to the direction of the search must be made at the time the rules are written. But, as we suggested above, this is often a useful thing to do anyway because it allows the rule writer to add control knowledge to the rules themselves.

CHAPTER

7

SYMBOLIC REASONING UNDER UNCERTAINTY

There are many methods for predicting the future. For example, you can read horoscopes, tea leaves, tarot cards, or crystal balls. Collectively, these methods are known as 'nutty methods.' Or you can put well-researched facts into sophisticated computer models, more commonly referred to as "a complete waste of time."

—Scott Adams
(1957-) Author Known for his comic strip Dilbert

So far, we have described techniques for reasoning with a complete, consistent, and unchanging model of the world. Unfortunately, in many problem domains it is not possible to create such models. In this chapter and the next, we explore techniques for solving problems with incomplete and uncertain models.

7.1 INTRODUCTION TO NONMONOTONIC REASONING

In their book, *The Web of Belief*, Quine and Ullian [1978] provide an excellent discussion of techniques that can be used to reason effectively even when a complete, consistent, and constant model of the world is not available. One of their examples, which we call the ABC Murder story, clearly illustrates many of the main issues that such techniques must deal with. Quoting Quine and Ullian [1978]:

Let Abbott, Babbitt, and Cabot be suspects in a murder case. Abbott has an alibi, in the register of a respectable hotel in Albany. Babbitt also has an alibi, for his brother-in-law testified that Babbitt was visiting him in Brooklyn at the time. Cabot pleads alibi too, claiming to have been watching a ski meet in the Catskills, but we have only his word for that. So we believe

1. That Abbott did not commit the crime
2. That Babbitt did not commit the crime
3. That Abbott or Babbitt or Cabot did.

But presently Cabot documents his alibi—he had the good luck to have been caught by television in the sidelines at the ski meet. A new belief is thus thrust upon us:

4. That Cabot did not.

Our beliefs (1) through (4) are inconsistent, so we must choose one for rejection. Which has the weakest evidence? The basis for (1) in the hotel register is good, since it is a fine old hotel. The basis for (2) is weaker, since Babbitt's brother-in-law might be lying. The basis for (3) is perhaps twofold: that there is no sign of burglary and that only Abbott, Babbitt, and Cabot seem to have stood to gain from the murder apart from burglary. This exclusion of burglary seems conclusive, but the other consideration does not; there could be some fourth beneficiary. For (4), finally, the basis is conclusive: the evidence from television. Thus (2) and (3) are the weak points. To resolve the inconsistency of (1) through (4) we should reject (2) or (3), thus either incriminating Babbitt or widening our net for some new suspect.

See also how the revision progresses downward. If we reject (2), we also revise our previous underlying belief, however tentative, that the brother-in-law was telling the truth and Babbitt was in Brooklyn. If instead we reject (3), we also revise our previous underlying belief that none but Abbott, Babbitt, and Cabot stood to gain from the murder apart from burglary.

Finally, a certain arbitrariness should be noted in the organization of this analysis. The inconsistent beliefs (1) through (4) were singled out, and then various further beliefs were accorded a subordinate status as underlying evidence: a belief about a hotel register, a belief about the prestige of the hotel, a belief about the television, a perhaps unwarranted belief about the veracity of the brother-in-law, and so on. We could instead have listed this full dozen of beliefs on an equal footing, appreciated that they were in contradiction, and proceeded to restore consistency by weeding them out in various ways. But the organization lightened our task. It focused our attention on four prominent beliefs among which to drop one, and then it ranged the other beliefs under these four as mere aids to choosing which of the four to drop.

The strategy illustrated would seem in general to be a good one: divide and conquer. When a set of beliefs has accumulated to the point of contradiction, find the smallest selection of them you can that still involves contradiction; for instance, (1) through (4). For we can be sure that we are going to have to drop some of the beliefs in that subset, whatever else we do. In reviewing and comparing the evidence for the beliefs in the subset, then, we will find ourselves led down in a rather systematic way to other beliefs of the set. Eventually we find ourselves dropping some of them too.

In probing the evidence, where do we stop? In probing the evidence for (1) through (4) we dredged up various underlying beliefs, but we could have probed further, seeking evidence in turn for them. In practice, the probing stops when we are satisfied how best to restore consistency: which ones to discard among the beliefs we have canvassed.

This story illustrates some of the problems posed by uncertain, fuzzy, and often changing knowledge. A variety of logical frameworks and computational methods have been proposed for handling such problems. In this chapter and the next, we discuss two approaches:

- Nonmonotonic reasoning, in which the axioms and/or the rules of inference are extended to make it possible to reason with incomplete information. These systems preserve, however, the property that, at any given moment, a statement is either believed to be true, believed to be false, or not believed to be either.
- Statistical reasoning, in which the representation is extended to allow some kind of numeric measure of certainty (rather than simply TRUE or FALSE) to be associated with each statement.

Other approaches to these issues have also been proposed and used in systems. For example, it is sometimes the case that there is not a single knowledge base that captures the beliefs of all the agents involved in solving a problem. This would happen in our murder scenario if we were to attempt to model the reasoning of Abbott, Babbitt, and Cabot, as well as that of the police investigator. To be able to do this reasoning, we would require a technique for maintaining several parallel *belief spaces*, each of which would correspond to the beliefs of one agent. Such techniques are complicated by the fact that the belief spaces of the various agents, although not identical, are sufficiently similar that it is unacceptably inefficient to represent them as completely separate knowledge bases. In Section 15.4.2 we return briefly to this issue. Meanwhile, in the rest of this chapter, we describe techniques for nonmonotonic reasoning.

Conventiotnal reasoning systems, such first-order predicate logic, are designed to work with information that has three important properties:

- It is complete with respect to the domain of interest. In other words, all the facts that are necessary to solve a problem are present in the system or can be derived from those that are by the conventional rules of first-order logic.
- It is consistent.
- The only way it can change is that new facts can be added as they become available. If these new facts are consistent with all the other facts that have already been asserted, then nothing will ever be retracted from the set of facts that are known to be true. This property is called *monotonicity.*

Unfortunately, if any of these properties is not satisfied, conventional logic-based reasoning systems become inadequate. Nonmonotonic reasoning systems, on the other hand, are designed to be able to solve problems in which all of these properties may be missing.

In order to do this, we must address several key issues, including the following:

1. *How can the knowledge base be extended to allow inferences to be made on the basis of lack of knowledge as well as on the presence of it?* For example, we would like to be able to say things like, "If you have no reason to suspect that a particular person committed a crime, then assume he didn't," or "If you have no reason to believe that someone is not getting along with her relatives, then assume that the relatives will try to protect her." Specifically, we need to make clear the distinction between:
   - It is known that $\neg P$.
   - It is not known whether P.

   First-order predicate logic allows reasoning to be based on the first of these. We need an extended system that allows reasoning to be based on the second as well. In our new system, we call any inference that depends on the lack of some piece of knowledge a *nonmonotonic inference.*[1]

   Allowing such reasoning has a significant impact on a knowledge base. Nonmonotonic reasoning systems derive their name from the fact that because of inferences that depend on lack of knowledge, knowledge bases may not grow monotonically as new assertions are made. Adding a new assertion may invalidate an inference that depended on the absence of that assertion. First-order predicate logic systems, on the other hand, are monotonic in this respect. As new axioms are asserted, new wff's may become provable, but no old proofs ever become invalid.

   In other words, if some set of axioms T entails the truth of some statement w, then T combined with another set of axioms N also entails w. Because nonmonotonic reasoning does not share this property, it is also called *defeasible:* a nonmonotonic inference may be defeated (rendered invalid) by the addition of new information that violates assumptions that were made during the original reasoning process. It turns out, as we show below, that making this one change has a dramatic impact on the structure of the logical system itself. In particular, most of our ideas of what it means to find a proof will have to be reevaluated.
2. *How can the knowledge base be updated properly when a new fact is added to the system (or when an old one is removed)?* In particular, in nonmonotonic systems, since the addition of a fact can cause

[1] Recall that in Section 2.4, we also made a monotonic/nonmonotonic distinction. There the issue was classes of production systems. Although we are applying the distinction to different entities here, it is essentially the same distinction in both cases, since it distinguishes between systems that never shrink as a result of an action (monotonic ones) and ones that can (nonmonotonic ones).

previously discovered proofs to be become invalid, how can those proofs, and all the conclusions that depend on them be found? The usual solution to this problem is to keep track of proofs, which are often called *justifications*. This makes it possible to find all the justifications that depended on the absence of the new fact, and those proofs can be marked as invalid. Interestingly, such a recording mechanism also makes it possible to support conventional, monotonic reasoning in the case where axioms must occasionally be retracted to reflect changes in the world that is being modeled. For example, it may be the case that Abbott is in town this week and so is available to testify, but if we wait until next week, he may be out of town. As a result, when we discuss techniques for maintaining valid sets of justifications, we talk both about nonmonotonic reasoning and about monotonic reasoning in a changing world.

3. *How can knowledge be used to help resolve conflicts when there are several in consistent nonmonotonic inferences that could be drawn?* It turns out that when inferences can be based on the lack of knowledge as well as on its presence, contradictions are much more likely to occur than they were in conventional logical systems in which the only possible contradictions were those that depended on facts that were explicitly asserted to be true. In particular, in nonmonotonic systems, there are often portions of the knowledge base that are locally consistent but mutually (globally) inconsistent. As we show below, many techniques for reasoning nonmonotonically are able to define the alternatives that could be believed, but most of them provide no way to choose among the options when not all of them can be believed at once.

To do this, we require additional methods for resolving such conflicts in ways that are most appropriate for the particular problem that is being solved. For example, as soon as we conclude that Abbott, Babbitt, and Cabot all claim that they didn't commit a crime, yet we conclude that one of them must have since there's no one else who is believed to have had a motive, we have a contradiction, which we want to resolve in some particular way based on other knowledge that we have. In this case, for example, we choose to resolve the conflict by finding the person with the weakest alibi and believing that he committed the crime (which involves believing other things, such as that the chosen suspect lied).

The rest of this chapter is divided into five parts. In the first, we present several logical formalisms that provide mechanisms for performing nonmonotonic reasoning. In the last four, we discuss approaches to the implementation of such reasoning in problem-solving programs. For more detailed descriptions of many of these systems, see the papers in Ginsberg [1987].

7.2 LOGICS FOR NONMONOTONIC REASONING

Because monotonicity is fundamental to the definition of first-order predicate logic, we are forced to find some alternative to support nonmonotonic reasoning. In this section, we look at several formal approaches to doing this. We examine several because no single formalism with all the desired properties has yet emerged (although there are some attempts, e.g., Shoham [1987] and Konolige [1987], to present a unifying framework for these several theories). In particular, we would like to find a formalism that does all of the following things:

- Defines the set of possible worlds that could exist given the facts that we do have. More precisely, we will define an *interpretation* of a set of wff's to be a domain (a set of objects) *D*, together with a function that assigns: to each predicate, a relation (of corresponding arity); to each n-ary function, an operator that maps from *D*" into *D;* and to each constant, an element of *D*. A *model* of a set of wff's is an interpretation that satisfies them. Now we can be more precise about this requirement. We require a mechanism for defining the set of models of any set of wff's we are given.
- Provides a way to say that we prefer to believe in some models rather than others.

- Provides the basis for a practical implementation of this kind of reasoning.
- Corresponds to our intuitions about how this kind of reasoning works. In other words, we do not want vagaries of syntax to have a significant impact on the conclusions that can be drawn within our system.

As we examine each of the theories below, we need to evaluate how well they perform each of these tasks. For a more detailed discussion of these theories and some comparisons among them, see Reiter [1987a], Etherington [1988], and Genesereth and Nilsson[1987].

Before we go into specific theories in detail, let's consider Fig. 7.1, which shows one way of visualizing how nonmonotonic reasoning works in all of them. The box labeled *A* corresponds to an original set of wff's. The large circle contains all the models of *A*. When we add some nonmonotonic reasoning capabilities to *A*, we get a new set of wff's, which we've labeled *B*.[2] *B* (usually) contains more information than *A* does. As a result, fewer models satisfy 5 than *A*. The set of models corresponding to *B* is shown at the lower right of the large circle. Now suppose we add some new wff's (representing new information) to *A*. We represent *A* with these additions as the box *C*. A difficulty may arise, however, if the set of models corresponding to *C* is as shown in the smaller, interior circle, since it is disjoint with the models for *B*. In order to find a new set of models that satisfy *C*, we need to accept models that had previously been rejected. To do that, we need to eliminate the wff's that were responsible for those models being thrown away. This is the essence of nonmonotonic reasoning.

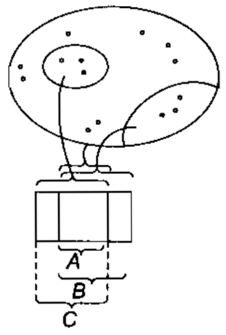


Fig. 7.1 *Models, Wff's, and Nonmonotonic Reasoning*

7.2.1 Default Reasoning

We want to use nonmonotonic reasoning to perform what is commonly called *default reasoning*. We want to draw conclusions based on what is most likely to be true. In this section, we discuss two approaches to doing this.

- Nonmonotonic Logic[3]
- Default Logic

We then describe two common kinds of nonmonotonic reasoning that can be defined in those logics:

- Abduction
- Inheritance

Nonmonotonic Logic

One system that provides a basis for default reasoning is *Nonmonotonic Logic* (NML) [McDermott and Doyle, 1980], in which the language of first-order predicate logic is augmented with a modal operator M, which can be read as "is consistent." For example, the formula

$$\forall x, y : Related(x, y) \wedge \mathrm{M}\ GetAlong(x, y) \rightarrow \neg WillDefend(x, y)$$

should be read as, "For all *x* and *y*, if *x* and *y* are related and if the fact that *x* gets along with *y* is consistent with everything else that is believed, then conclude that *x* will defend *y*."

[2] As we will see below, some techniques add inference rules, which then generate wff's, while others add wff's directly. We'll ignore that difference for the moment.

[3] Try not to get confused about names here. We are using the terms "nonmonotonic reasoning" and "default reasoning" generically to describe a kind of reasoning. The terms "Nonmonotonic Logic" and "Default Logic" are, on the other hand, being used to refer to specific formal theories.

Once we augment our theory to allow statements of this form, one important issue must be resolved if we want our theory to be even semidecidable. (Recall that even in a standard first-order theory, the question of theoremhood is undecidable, so semide-cidability is the best we can hope for.) We must define what "is consistent" means. Because consistency in this system, as in first-order predicate logic, is undecidable, we need some approximation. The one that is usually used is the PROLOG notion of negation as failure, or some variant of it. In other words, to show that P is consistent, we attempt to prove ¬P. If we fail, then we assume ¬ to be false and we call P consistent. Unfortunately, this definition does not completely solve our problem. Negation as failure works in pure PROLOG because, if we restrict the rest of our language to Horn clauses, we have a decidable theory. So failure to prove something means that it is not entailed by our theory. If, on the other hand, we start with full first-order predicate logic as our base language, we have no such guarantee. So, as a practical matter, it may be necessary to define consistency on some heuristic basis, such as failure to prove inconsistency within some fixed level of effort.

A second problem that arises in this approach (and others, as we explain below) is what to do when multiple nonmonotonic statements, taken alone, suggest ways of augmenting our knowledge that if taken together would be inconsistent. For example, consider the following set of assertions:

$\forall x : Republican(x) \wedge \mathrm{M}\ \neg Pacifist(x) \rightarrow \neg Pacifist(x)$
$\forall x : Quaker(x) \wedge \mathrm{M}\ Pacifist(x) \rightarrow Pacifist(x)$
$Republican(Dick)$
$Quakev\{Dick)$

The definition of NML that we have given supports two distinct ways of augmenting this knowledge base. In one, we first apply the first assertion, which allows us to conclude ¬*Pacifist(Dick)*. Having done that, the second assertion cannot apply, since it is not consistent to assume *Pacifist(Dick)*. The other thing we could do, however, is apply the second assertion first. This results in the conclusion *Pacifist(Dick),* which prevents the first one from applying. So what conclusion does the theory actually support?

The answer is that NML defines the set of theorems that can be derived from a set of wff's A to be the intersection of the sets of theorems that result from the various ways in which the wff's of A might be combined. So, in our example, no conclusion about Dick's pacifism can be derived. This theory thus takes a very conservative approach to theoremhood.

It is worth pointing out here that although assertions such as the ones we used to reason about Dick's pacifism look like rules, they are, in this theory, just ordinary wff's which can be manipulated by the standard rules for combining logical expressions. So, for example, given

$A \wedge \mathrm{M}\ B \rightarrow B$
$\neg A \wedge \mathrm{M}\ B \rightarrow B$

we can derive the expression

$\mathrm{M}\ B \rightarrow B$

In the original formulation of NML, the semantics of the modal operator M, which is self-referential, were unclear. A more recent system, *Autoepistemic Logic* [Moore, 1985] is very similar, but solves some of these problems.

[4] Reiter's original notation had ":M" in place of ":". but since it conveys no additional information, the M is usually omitted.

Default Logic

An alternative logic for performing default-based reasoning is Reiter's *Default Logic* (DL) [Reiter, 1980], in which a new class of inference rules is introduced. In this approach, we allow inference rules of the form[4]

$$\frac{A : B}{C}$$

Such a rule should be read as, "If A is provable and it is consistent to assume B then conclude C." As you can see, this is very similar in intent to the nonmonotonic expressions that we used in NML. There are some important differences between the two theories, however. The first is that in DL the new inference rules are used as a basis for computing a set of plausible *extensions* to the knowledge base. Each extension corresponds to one maximal consistent augmentation of the knowledge base.[5] The logic then admits as a theorem any expression that is valid in any extension. If a decision among the extensions is necessary to support problem solving, some other mechanism must be provided. So, for example, if we return to the case of Dick the Republican, we can compute two extensions, one corresponding to his being a pacifist and one corresponding to his not being a pacifist. The theory of DL does not say anything about how to choose between the two. But see Reiter and Criscuolo [1981], Touretzky [1986], and Rich [1983] for discussions of this issue.

A second important difference between these two theories is that, in DL, the nonmonotonic expressions are rules of inference rather than expressions in the language. Thus they cannot be manipulated by the other rules of inference. This leads to some unexpected results. For example, given the two rules

$$\frac{A : B}{C} \qquad \frac{\neg A : B}{B}$$

and no assertion about A, no conclusion about B will be drawn, since neither inference rule applies.

Abduction

Standard logic performs deduction. Given two axioms:

$$\forall x : A(x) \rightarrow B(x)$$
$$A(C)$$

we can conclude $B(C)$ using deduction. But what about applying the implication in reverse? For example, suppose the axiom we have is.

$$\forall x : Measles(x) \rightarrow Spots(x)$$

The axiom says that having measles implies having spots. But suppose we notice spots. We might like to conclude measles. Such a conclusion is not licensed by the rules of standard logic and it may be wrong, but it may be the best guess we can make about what is going on. Deriving conclusions in this way is thus another form of default reasoning. We call this specific form *abductive reasoning*. More precisely, the process of abductive reasoning can be described as, "Given two wff's ($A \rightarrow B$) and (B), for any expressions A and B, if it is consistent to assume A, do so."

In many domains, abductive reasoning is particularly useful if some measure of certainty is attached to the resulting expressions. These certainty measures quantify the risk that the abductive reasoning process is

[5] What we mean by the expression "maximal consistent augmentation" is that no additional default rules can be applied without violating consistency. But its is important to note that only expressions generated by the application of the stated inference rules to the original knowledge are allowed in an extension. Gratuitous additions are not permitted.

wrong, which it will be whenever there were other antecedents besides *A* that could have produced *B*. We discuss ways of doing this in Chapter 8.

Abductive reasoning is not a kind of logic in the sense that DL and NML are. In fact, it can be described in either of them. But it is a very useful kind of nonmonotonic reasoning, and so we mentioned it explicitly here.

Inheritance

One very common use of nonmonotonic reasoning is as a basis for inheriting attribute values from a prototype description of a class to the individual entities that belong to the class. We considered one example of this kind of reasoning in Chapter 4, when we discussed the baseball knowledge base. Recall that we presented there an algorithm for implementing inheritance. We can describe informally what that algorithm does by saying, "An object inherits attribute values from all the classes of which it is a member unless doing so leads to a contradiction, in which case a value from a more restricted class has precedence over a value from a broader class." Can the logical ideas we have just been discussing provide a basis for describing this idea more formally? The answer is yes. To see how, let's return to the baseball example (as shown in Figure 4.5) and try to write its inheritable knowledge as rules in DL.

We can write a rule to account for the inheritance of a default value for the height of a baseball player as:

$$\frac{\textit{Baseball-Player}(x) : \textit{height}(x, \text{6-1})}{\textit{height}(x, \text{6-1})}$$

Now suppose we assert *Pitcher(Three-Finger-Brown)*. Since this enables us to conclude that *Three-Finger-Brown* is a baseball player, our rule allows us to conclude that his height is 6-1. If, on the other hand, we had asserted a conflicting value for Three Finger' had an axiom like

$$\forall x, y, z : \textit{height}(x, y) \wedge \textit{height}(x, z) \rightarrow y = z,$$

which prohibits someone from having more than one height, then we would not be able to apply the default rule. Thus an explicitly stated value will block the inheritance of a default value, which is exactly what we want. (We'll ignore here the order in which the assertions and the rules occur. As a logical framework, default logic does not care. We'll just assume that somehow it settles out to a consistent state in which no defaults that conflict with explicit assertions have been asserted. In Section 7.5.1 we look at issues that arise in creating an implementation that assures that.)

But now, let's encode the default rule for the height of adult males in general. If we pattern it after the one for baseball players, we get

$$\frac{\textit{Adult-Male}(x) : \textit{height}(x, \text{5-10})}{\textit{height}(x, \text{5-10})}$$

Unfortunately, this rule does not work as we would like. In particular, if we again assert *Pitcher(Three-Finger-Brown),* then the resulting theory contains two extensions: one in which our first rule fires and Brown's height is 6-1 and one in which this new rule applies and Brown's height is 5-10. Neither of these extensions is preferred. In order to state that we prefer to get a value from the more specific category, baseball player, we could rewrite the default rule for adult males in general as:

$$\frac{\textit{Adult-Male}(x) : \neg \textit{Baseball-Player}(x) \wedge \textit{height}(x, \text{5-10})}{\textit{height}(x, \text{5-10})}$$

This effectively blocks the application of the default knowledge about adult males in the case that more specific information from the class of baseball players is available.

Unfortunately, this approach can become unwieldy as the set of exceptions to the general rule increases. For example, we could end up with a rule like:

$$\frac{Adult\text{-}Male(x) : \neg Baseball\text{-}Player(x) \wedge \neg Midget(x) \wedge - Jockey(x) \wedge height(x, 5\text{-}10)}{height(x, 5\text{-}10)}$$

What we have done here is to clutter our knowledge about the general class of adult males with a list of all the known exceptions with respect to height. A clearer approach is to say something like, "Adult males typically have a height of 5-10 unless they are abnormal in some way." We can then associate with other classes the information that they are abnormal in one or another way. So we could write, for example:

$\forall x$: $Adult\text{-}Male(x) \wedge \neg AB(x, aspect1) \rightarrow height(x, 5\text{-}10)$
$\forall x$: $Baseball\text{-}Player(x) \rightarrow AB(x, aspect\ 1)$
$\forall x$: $Midget(x) \rightarrow AB(x, aspect\ 1)$
$\forall x$: $Jockey(x) \rightarrow AB(x, aspect\ 1)$

Then, if we add the single default rule:

$$\frac{: \neg AB(x,\ y)}{\neg AB(x,\ y)}$$

we get the desired result.

7.2.2 Minimalist Reasoning

So far, we have talked about general methods that provide ways of describing things that are generally true. In this section we describe methods for saying a very specific and highly useful class of things that are generally true. These methods are based on some variant of the idea of a *minimal model.* Recall from the beginning of this section that a model of a set of formulas is an interpretation that satisfies them. Although there are several distinct definitions of what constitutes a minimal model, for our purposes, we will define a model to be minimal if there are no other models in which fewer things are true. (As you can probably imagine, there are technical difficulties in making this precise, many of which involve the treatment of sentences with negation.) The idea behind using minimal models as a basis for nonmonotonic reasoning about the world is the following: "There are many fewer true statements than false ones. If something is true and relevant it makes sense to assume that it has been entered into our knowledge base. Therefore, assume that the only true statements are those that necessarily must be true in order to maintain the consistency of the knowledge base." We have already mentioned (in Section 6.2) one kind of reasoning based on this idea, the PROLOG concept of negation as failure, which provides an implementation of the idea for Horn clause-based systems. In the rest of this section we look at some logical issues that arise when we remove the Horn clause limitation.

The Closed World Assumption

A simple kind of minimalist reasoning is suggested by the *Closed World Assumption* or CWA [Reiter, 1978]. The CWA says that the only objects that satisfy any predicate *P* are those that must. The CWA is particularly powerful as a basis for reasoning with databases, which are assumed to be complete with respect to the properties they describe. For example, a personnel database can safely be assumed to list all of the company's employees. If someone asks whether Smith works for the company, we should reply "no" unless he is explicitly listed as an employee. Similarly, an airline database can be assumed to contain a complete list of all the routes flown by that airline. So if I ask if there is a direct flight from Oshkosh to El Paso, the answer should be "no" if none can be found in the database. The CWA is also useful as a way to deal with *AB* predicates, of the sort we introduced in Section 7.2.1, since we want to take as abnormal only those things that are asserted to be so.

Although the CWA is both simple and powerful, it can fail to produce an appropriate answer for either of two reasons. The first is that its assumptions are not always true in the world; some parts of the world are not realistically "closable." We saw this problem in the murder story example. There were facts that were relevant to the investigation that had not yet been uncovered and so were not present in the knowledge base. The CWA will yield appropriate results exactly to the extent that the assumption that all the relevant positive facts are present in the knowledge base is true.

The second kind of problem that plagues the CWA arises from the fact that it is a purely syntactic reasoning process. Thus, as you would expect, its results depend on the form of the assertions that are provided. Let's look at two specific examples of this problem.

Consider a knowledge base that consists of just a single statement:

$A(Joe) \vee B(Joe)$

The CWA allows us to conclude both ? $A(Joe)$ and ?$B(Joe)$, since neither A nor 6 must necessarily be true of Joe. Unfortunately, the resulting extended knowledge base

$A(Joe) \vee B(Joe)$
$\neg A(Joe)$
$\neg B(Joe)$

is inconsistent.

The problem is that we have assigned a special status to positive instances of predicates, as opposed to negative ones. Specifically, the CWA forces completion of a knowledge base by adding the negative assertion *P whenever it is consistent to do so. But the assignment of a real world property to some predicate P and its complement to the negation of P may be arbitrary. For example, suppose we define a predicate *Single* and create the following knowledge base:

$Single(John)$
$Single(Mary)$

Then, if we ask about Jane, the CWA will yield the answer $\neg Single(Jane)$. But now suppose we had chosen instead to use the predicate *Married* rather than *Single*. Then the corresponding knowledge base would be

$\neg Married(Johri)$
$\neg Married(Mary)$

If we now ask about Jane, the CWA will yield the result $\neg Married(Jane)$.

Circumscription

Although the CWA captures part of the idea that anything that must not necessarily be true should be assumed to be false, it does not capture all "of it. It has two essential limitations:

- It operates on individual'predicates without considering the interactions among predicates that are defined in the knowledge hase. We saw an example of this above when we considered the statement $A(Joe) \vee B(Joe)$.
- It assumes that all predicates have all of their instances listed. Although in many database applications this is true, in many knowledge-based systems it is not. Some predicates can reasonably be assumed to

be completely defined (i.e., the part of the world they describe is closed), but others cannot (i.e., the part of the world they describe is open). For example, the predicate *has-a-green-shirt* should probably be considered open since in most situations it would not be safe to assume that one has been told all the details of everyone else's wardrobe.

Several theories *of circumscription* (e.g., McCarthy [1980], McCarthy [1986], and Lifschitz [1985]) have been proposed to deal with these problems. In all of these theories, new axioms are added to the existing knowledge base. The effect of these axioms is to force a minimal interpretation on "a selected portion of the knowledge base. In particular, each specific axiom describes a way that the set of values for which a particular axiom of the original theory is true is to be delimited (i.e., circumscribed).

As an example, suppose we have the simple assertion

$$\forall x : Adult(x) \wedge \neg AB(x, aspect1) \rightarrow Literate(x)$$

We would like to circumscribe *AB*, since we would like it to apply only to those individuals to which it applies. In essence, what we want to do is to say something about what the predicate *AB* must be (since at this point we have no idea what it is; all we know is its name). To know what it is, we need to know for what values it is true. Even though we may know a few values for which it is true (if any individuals have been asserted to be abnormal in this way), there are many different predicates that would be consistent with what we know so far. Imagine this universe of possible binary predicates. We might ask, which of these predicates could be *AB?* We want to say that *AB* can only be one of the predicates that is true only for those objects that we know it must be true for. We can do this by adding a (second order) axiom that says that *AB* is the smallest predicate that is consistent with our existing knowledge base.

In this simple example, circumscription yields the same result as does the CWA since there are no other assertions in the knowledge base with which a minimization of *AB* must be consistent. In both cases, the only models that are admitted are ones in which there are no individuals who are abnormal in *aspect 1*. In other words, *AB* must be the predicate FALSE.

But, now let's return to the example knowledge base

$$A(Joe) \vee B(Joe)$$

If we circumscribe only *A*, then this assertion describes exactly those models in which A is true of no one and *B* is true of at least *Joe*. Similarly, if we circumscribe only *B*, then we will accept exactly those models in which *B* is true of no one and *A* is true of at least *Joe*. If we circumscribe *A* and *B* together, then we will admit only those models in which *A* is true of only *Joe* and *B* is true of no one or those in which B is true of only *Joe* and *A* is true of no one. Thus, unlike the CWA, circumscription allows us to describe the logical relationship between *A* and *B*.

7.3 IMPLEMENTATION ISSUES

Although the logical frameworks that we have just discussed take us part of the way toward a basis for implementing nonmonotonic reasoning in problem-solving programs, they are not enough. As we have seen, they all have some weaknesses as logical systems. In addition, they fail to deal with four important problems that arise in real systems.

The first is how to derive exactly those nonmonotonic conclusions that are relevant to solving the problem at hand while not wasting time on those that, while they may be licensed by the logic, are not necessary and are not worth spending time on.

We can apply the numerator of the combination rule to produce (where * denotes the empty set):

		$\{A\}$	(0.9)	Θ	(0.1)
$\{F, C\}$	(0.48)	ϕ	(0.432)	$\{F, C\}$	(0.048)
$\{A, F, C\}$	(0.32)	$\{A, F, C\}$	(0.288)	$\{A, F, C\}$	(0.032)
$\{F, C, P\}$	(0.12)	ϕ	(0.108)	$\{F, C, P\}$	(0.012)
Θ	(0.08)	$\{A\}$	(0.072)	Θ	(0.008)

But there is now a total belief of 0.54 associated with ϕ; only 0.45 is associated with outcomes that are in fact possible. So we need to scale the remaining values by the factor 1 — 0.54 = 0.46. If we do this, and also combine alternative ways of generating the set {*All, Flu, Cold*}, then we get the final combined belief function, m_5.

{*Flu, Cold*}	(0.104)
{*All, Flu, Cold*}	(0.696)
{*Flu, Cold, Pneu*}	(0.026)
{*All*}	(0.157)
Θ	(0.017)

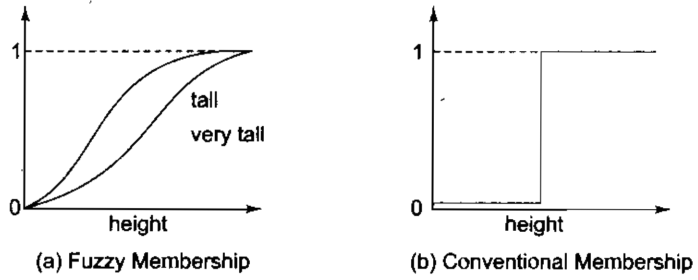


Fig. 8.4 *Fuzzy versus Conventional Set Membership*

In this example, the percentage of m_5 that was initially assigned to the empty set was large (over half). This happens whenever there is conflicting evidence (as in this case between m_1 and m_4).

8.5 FUZZY LOGIC

In the techniques we have discussed so far, we have hot modified the mathematical underpinnings provided by set theory and logic. We have instead augmented those ideas with additional constructs provided by probability theory. In this section, we take a different approach and briefly consider what happens if we make fundamental changes to our idea of set membership and corresponding changes to our definitions of logical operations.

The motivation for fuzzy sets is provided by the need to represent such propositions as:

John is very tall.
Mary is slightly ill.
Sue and Linda are close friends.
Exceptions to the rule are nearly impossible.
Most Frenchmen are not very tall.

While traditional set theory defines set membership as a boolean predicate, fuzzy set theory allows us to represent set membership as a possibility distribution, such as the ones shown in Fig. 8.4(*a*) for the set of tall people and the set of very tall people. Notice how this contrasts with the standard boolean definition for tall people shown in Fig. 8.4(b). In the latter, one is either tall or not and there must be a specific height that defines the boundary. The same is true for very tall. In the former, one's tallness increases with one's height until the value of 1 is reached.

Once set membership has been redefined in this way, it is possible to define a reasoning system based on techniques for combining distributions [Zadeh, 1979] (or see the papers in the journal *Fuzzy Sets and Systems*). Such reasoners have been applied in control systems for devices as diverse as trains and washing machines. A typical fuzzy logic control system has been described in Chapter 22.

SUMMARY

In this chapter we have shown that Bayesian statistics provide a good basis for reasoning under various kinds of uncertainty. We have also, though, talked about its weaknesses in complex real tasks, and so we have talked about ways in which it can be modified to work in practical domains. The thing that all of these modifications have in common is that they substitute, for the huge joint probability matrix that a pure Bayesian approach requires, a more structured representation of the facts that are relevant to a particular problem. They typically do this by combining probabilistic information with knowledge that is represented using one or more other representational mechanisms, such as rules or constraint networks.

Comparing these approaches for use in a particular problem-solving program is not always straightforward, since they differ along several dimensions, for example:

- They provide different mechanisms for describing the ways in which propositions are not independent of each other.
- They provide different techniques for representing ignorance.
- They differ substantially in the ease with which systems that use them can be built and in the computational complexity that the resulting systems exhibit.

We have also presented fuzzy logic as an alternative for representing some kinds of uncertain knowledge. Although there remain many arguments about the relative overall merits of the Bayesian and the fuzzy approaches, there is some evidence that they may both be useful in capturing different kinds of information. As an example, consider the proposition

John was pretty sure that Mary was seriously ill.

Bayesian approaches naturally capture John's degree of certainty, while fuzzy techniques ran describe the degree of Mary's illness.

Throughout all of this discussion, it is important to keep in mind the fact that although we have been discussing techniques for representing knowledge, there is another perspective from which what we have really been doing is describing ways of representing *lack* of knowledge. In this sense, the techniques we have described in this chapter are fundamentally different from the ones we talked about earlier. For example, the truth values that we manipulate in a logical system characterize the formulas that we write; certainty measures, on the other hand, describe the exceptions — the facts that do not appear anywhere in the formulas that we have written. The consequences of this distinction show up in the ways that we can interpret and manipulate the formulas that we write. The most important difference is that logical formulas can be treated as though they represent independent propositions. As we have seen throughout this chapter, uncertain assertions cannot. As a result, for example, while implication is transitive in logical systems, we often get into trouble in uncertain

CHAPTER

9

WEAK SLOT-AND-FILLER STRUCTURES

Speech is the representation of the mind, and writing is the representation of speech

—Aristotle
(384 BC – 322 BC), Greek philosopher

In this chapter, we continue the discussion we began in Chapter 4 of slot-and-filler structures. Recall that we originally introduced them as a device to support property inheritance along *isa* and *instance* links. This is an important aspect of these structures. Monotonic inheritance can be performed substantially more efficiently with such structures than with pure logic, and nonmonotonic inheritance is easily supported. The reason that inheritance is easy is that the knowledge in slot-and-filler systems is structured as a set of entities and their attributes. This structure turns out to be a useful one for other reasons besides the support of inheritance, though, including:

- It indexes assertions by the entities they describe. More formally, it indexes binary predicates [such as *team*{*Three-Finger-Brown, Chicago-Cubs*) by their first argument. As a result, retrieving the value for an attribute of an entity is fast.
- It makes it easy to describe properties of relations. To do this in a purely logical system requires some higher-order mechanisms.
- It is a form of object-oriented programming and has the advantages that such systems normally have, including modularity and ease of viewing by people.

We describe two views of this kind of structure: semantic nets and frames. We talk about the representations themselves and about techniques for reasoning with them. We do not say much, though, about the specific knowledge that the structures should contain. We call these "knowledge-poor" structures "weak," by analogy with the weak methods for problem solving that we discussed in Chapter 3. In the next chapter, we expand this discussion to include "strong" slot-and-filler structures, in which specific commitments to the content of the representation are made.

9.1 SEMANTIC NETS

The main idea behind semantic nets is that the meaning of a concept comes from the ways in which it is connected to other concepts. In a semantic net, information is represented as a set of nodes connected to each other by a set of labeled arcs, which represent relationships among the nodes. A fragment of a typical semantic net is shown in Fig. 9.1.

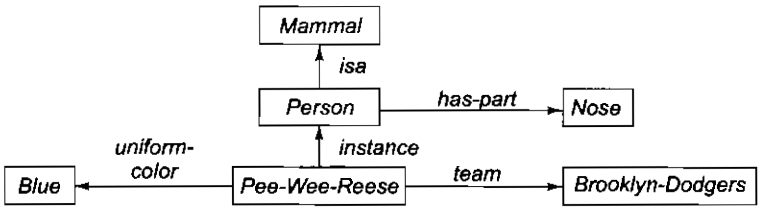


Fig. 9.1 *A Semantic Network*

This network contains examples of both the *isa* and *instance* relations, as well as some other, more domain-specific relations like *team* and *uniform-color*. In this network, we could use inheritance to derive the additional relation

has-part (Pee-Wee-Reese, Nose)

9.1.1 Intersection Search

One of the early ways that semantic nets were used was to find relationships among objects by spreading activation out from each of two nodes and seeing where the activation met. This process is called *intersection search* [Quillian, 1968]. Using this process, it is possible to use the network of Fig. 9.1 to answer questions such as "What is the connection between the Brooklyn Dodgers and blue?"[1] This kind of reasoning exploits one of the important advantages that slot-and-filler structures have over purely logical representations because it takes advantage of the entity-based organization of knowledge that slot-and-filer representations provide.

To answer more structured questions, however, requires networks that are themselves more highly structured. In the next few sections we expand and refine our notion of a network in order to support more sophisticated reasoning.

9.1.2 Representing Nonbinary Predicates

Semantic nets are a natural way to represent relationships that would appear as ground instances of binary predicates in predicate logic. For example, some of the arcs from Fig. 9.1 could be represented in logic as

isa(Person, Mammal)
instance(Pee-Wee-Reese, Person)
team(Pee-Wee-Reese, Brooklyn-Dodgers)
uniform-color(Pee-Wee-Reese, Blue)

But the knowledge expressed by predicates of other arities can also be expressed in semantic nets. We have already seen that many unary predicates in logic can be thought of as binary predicates using some very general-purpose predicates, such as *isa* and *instance*. So, for example,

man(Marcus)

[1] Actually, to do this we need to assume that the inverses of the links we have shown also exist.

could be rewritten as

instance(*Marcus, Man*)

thereby making it easy to represent in a semantic net.

Three or more place predicates can also be converted to a binary form by creating one new object representing the entire predicate statement and then introducing binary predicates to describe the relationship to this new object of each of the original arguments. For example, suppose we know that

score(*Cubs, Dodgers,* 5-3)

This can be represented in a semantic net by creating a node to represent the specific game and then relating each of the three pieces of information to it. Doing this produces the network shown in Fig. 9.2.

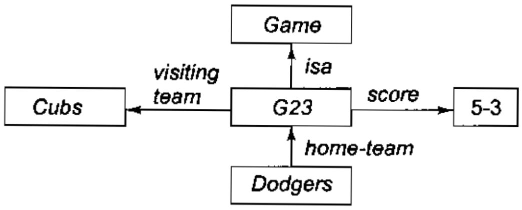


Fig. 9.2 *A Semantic Net for an n-Place Predicate*

This technique is particularly useful for representing the contents of a typical declarative sentence that describes several aspects of a particular event. The sentence

John gave the book to Mary.

could be represented by the network shown in Fig. 9.3.[2] In fact, several of the earliest uses of semantic nets were in English-understanding programs.

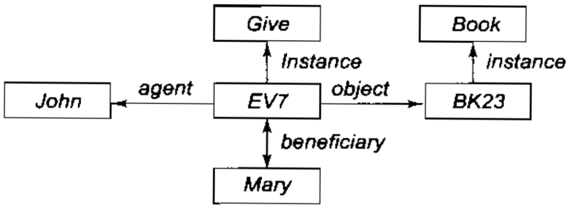
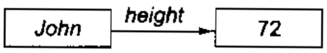


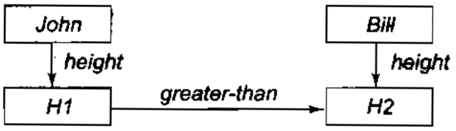
Fig. 9.3 *A Semantic Net Representing a Sentence*

9.1.3 Making Some Important Distinctions

In the networks we have described so far, we have glossed over some distinctions that are important in reasoning. For example, there should be a difference between a link that defines a new entity and one that relates two existing entities. Consider the net



Both nodes represent objects that exist independently of their relationship to each other. But now suppose we want to represent the fact that John is taller than Bill, using the net

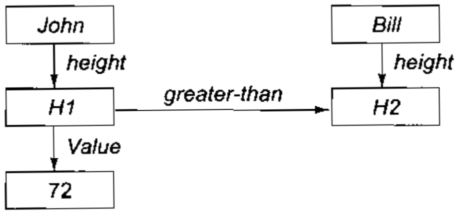


The nodes *H*1 and *H*2 are new concepts representing John's height and Bill's height, respectively. They are defined by their relationships to the nodes *John* and *Bill.* Using these defined concepts, it is possible to

[2] The node labeled *BK23* represents the particular book that was referred to by the phrase "the book." Discovering which particular book was meant by that phrase is similar to the problem of deciding on the correct referent for a pronoun, and it can be a very hard problem. These issues are discussed in Section 15.4.

represent such facts as that John's height increased, which we could not do before. (The number 72 increased?)

Sometimes it is useful to introduce the arc *value* to make this distinction clear. Thus we might use the following net to represent the fact that John is 6 feet tall and that he is taller than Bill:



The procedures that operate on nets such as this can exploit the fact that some arcs, such as *height,* define new entities, while others, such as *greater-than* and *value,* merely describe relationships among existing entities.

Another example of an important distinction we have missed is the difference between the properties of a node itself and the properties that a node simply holds and passes on to its instances. For example, it is a property of the node *Person* that it is a subclass of the node *Mammal.* But the node *Person* does not have as one of its parts a nose. Instances of the node *Person* do, and we want them to inherit it.

It is difficult to capture these distinctions without assigning more structure to our notions of node, link, and value. In the next section, when we talk about frame systems, we do that. But first, we discuss a network-oriented solution to a simpler problem; this solution illustrates what can be done in the network model but at what price in complexity.

9.1.4 Partitioned Semantic Nets

Suppose we want to represent simple quantified expressions in semantic nets. One way to do this is to *partition* the semantic net into a hierarchical set of *spaces,* each of which corresponds to the scope of one or more variables [Hendrix, 1977]. To see how this works, consider first the simple net shown in Fig. 9.4(a). This net corresponds to the statement

The dog bit the mail carrier

The nodes *Dot's, Bite,* and *Mail-Carrier* represent the classes of dogs, bitings, and mail carriers, respectively, while the nodes *d, b,* and *m* represent a particular dog, a particular biting, and a particular mail carrier. This fact can easily be represented by a single net with no partitioning.

But now suppose that we want to represent the fact

Every dog has bitten a mail carrier,

or, in logic:

$$\forall x : Dog(x) \rightarrow \exists y : Mail\text{-}Carrier\ (y) \wedge Bite \wedge (x, y)$$

To represent this fact, it is necessary to encode the scope of the universally quantified variable *x*. This can be done using partitioning as shown in Fig. 9.4(b). The node *g* stands for the assertion given above. Node *g* is an instance of the special class *GS* of general statements about the world (i.e., those with universal quantifiers). Every element of *GS* has at least two attributes: a *form,* which states the relation that is being asserted, and one or more ∀ connections, one for each of the universally quantified variables. In this example, there is only one such variable *d*, which can stand for any element of the class *Dogs*. The other two variables in the form, *b* and *m*, are understood to be existentially quantified. In other words, for every dog *d*, there exists a biting event *b*, and a mail carrier *m*, such that *d* is the assailant of *b* and *m* is the victim.

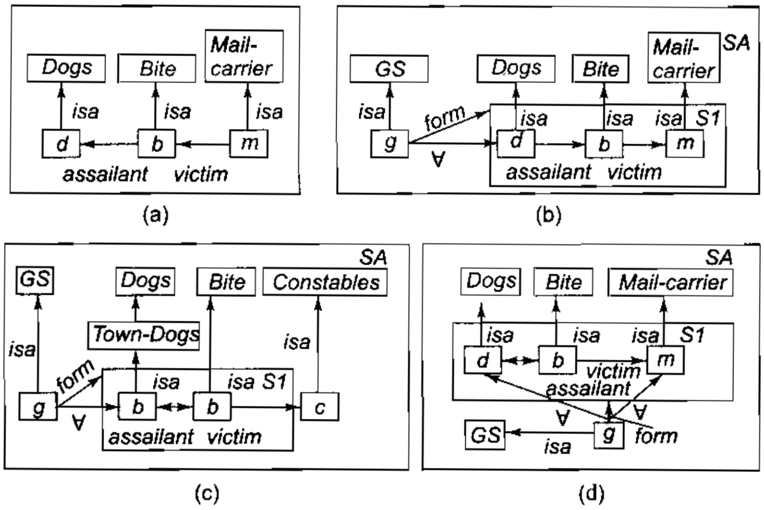


Fig. 9.4 *Using Partitioned Semantic Nets*

To see how partitioning makes variable quantification explicit, consider next the similar sentence:

Every dog in town has bitten the constable.

The representation of this sentence is shown in Fig. 9.4(c). In this net, the node *c* representing the victim lies outside the form of the general statement. Thus it is not viewed as an existentially quantified variable whose value may depend on the value of *d*. Instead it is interpreted as standing for a specific entity (in this case, a particular constable), just as do other nodes in a standard, nonpartitioned net.

Figure 9.4(d) shows how yet another similar sentence:

Every dog has bitten every mail carrier.

would be represented. In this case, *g* has two ∀ links, one pointing to *d*, which represents any dog, and one pointing to *m*, representing any mail carrier.

The spaces of a partitioned semantic net are related to each other by an inclusion hierarchy. For example, in Fig. 9.4(d), space *S*1 is included in space *SA*. Whenever a search process operates in a partitioned semantic net, it can explore nodes and arcs in the space from which it starts and in other spaces that contain the starting point, but it cannot go downward, except in special circumstances, such as when *a form* arc is being traversed. So, returning to Fig. 9.4(d), from node *d* it can be determined that *d* must be a dog. But if we were to start at the node *Dogs* and search for all known instances of dogs by traversing *isa* links, we would not find *d* since it and the link to it are in the space *S*1, which is at a lower level than space *SA*, which contains *Dogs*. This is important, since *d* does not stand for a particular dog; it is merely a variable that can be instantiated with a value that represents a dog.

9.1.5 The Evolution into Frames

The idea of a semantic net started out simply as a way to represent labeled connections among entities. But, as we have just seen, as we expand the range of problem-solving tasks that the representation must support, the representation itself necessarily begins to become more complex. In particular, it becomes useful to assign more structure to nodes as well as to links. Although there is no clear distinction between a semantic net and a frame system, the more structure the system has, the more likely it is to be termed a frame system. In the next section we continue our discussion of structured slot-and-filler representations by describing some of the most important capabilities that frame systems offer.

9.2 FRAMES

A frame is a collection of attributes (usually called slots) and associated values (and possibly constraints on values) that describe some entity in the world. Sometimes a frame describes an entity in some absolute sense; sometimes it represents the entity from a particular point of view (as it did in the vision system proposal [Minsky, 1975] in which the term *frame* was first introduced). A single frame taken alone is rarely useful. Instead, we build frame systems out of collections of frames that are connected to each other by virtue of the fact that tbe value of an attribute of one frame may be another frame. In the rest of this section, we expand on this simple definition and explore ways that frame systems can be used to encode knowledge and support reasoning

9.2.1 Frames as Sets and Instances

The Set theory provides a good basis for understanding frame systems. Although not all frame systems are defined this way, we do so here. In this view, each frame represents either a class (a set) or an instance (an element of a class). To see how this works, consider the frame system shown in Fig. 9.5, which is a slightly modified form of the network we showed in Fig. 9.5. In this example, the frames *Person, Adult-Male, ML-Baseball-Player* (corresponding to major league baseball players), *Pitcher*, and *ML-Baseball-Team* (for major league baseball team) are all classes. The frames *Pee-Wee-Reese* and *Brooklyn-Dodgers* are instances.

The *isa* relation that we have been using without a precise definition is in fact the *subset* relation. The set of adult males is a subset of the set of people. The set of major league baseball players is a subset of the set of adult males, and so forth. Our *instance* relation corresponds to the relation *element-of*. Pee Wee Reese is an element of the set of fielders. Thus he is also an element of all of the supersets of fielders, including major league baseball players and people. The transitivity of *isa* that we have taken for granted in our description of property inheritance follows directly from the transitivity of the subset relation.

Both the *isa* and *instance* relations have inverse attributes, which we call *subclasses* and *all-instances*. We do not bother to write them explicitly in our examples unless we need to refer to them. We assume that the frame system maintains them automatically, either explicitly or by computing them if necessary.

Because a class represents a set, there are two kinds of attributes that can be associated with it. There are attributes about the set itself, and there are attributes that are to be inherited by each element of the set. We indicate the difference between these two by prefixing the latter with an asterisk (*). For example, consider the class *ML-Baseball-Player*, We have shown only two properties of it as a set: It is a subset of the set of adult males. And it has cardinality 624 (i.e., there are 624 major league baseball players). We have listed five properties that all major league baseball players have (*height, bats, batting-average, team,* and *uniform-color*), and we have specified default values for the first three of them. By providing both kinds of slots, we allow a class both to define a set of objects and to describe a prototypical object of the set.

Sometimes, the distinction between a set and an individual instance may not seem clear. For example, the team *Brooklyn-Dodgers*, which we have described as an instance of the class of major league baseball teams, could be thought of as a set of players. In fact, notice that the value of the slot *players* is a set. Suppose, instead, that we want to represent the Dodgers as a class instead of an instance. Then its instances would be the individual players. It cannot stay where it is in the *isa* hierarchy; it cannot be a subclass of *ML-Baseball-Team,* because if it were, then its elements, namely the players, would also, by the transitivity of subclass, be elements of *ML-Baseball-Team,* which is not what we want to say. We have to put it somewhere else in the *isa* hierarchy. For example, we could make it a subclass of major league baseball players. Then its elements, the players, are also elements of *ML-Baseball-Player, Adult-Male,* and *Person.* That is acceptable. But if we do that, we lose the ability to inherit properties of the Dodgers from general information about baseball teams. We can still inherit attributes for the elements of the team, but we cannot inherit properties of the team as a whole, i.e., of the set of players. For example, we might like to know what the default size of the team is,

```
Person
    isa :                 Mammal
    cardinality :         6,000,000,000
    * handed :            Right
Adult-Male
    isa :                 Person
    cardinality :         2,000,000,000
    * height :            5-10
ML-Baseball-Player
    isa :                 Adult-Male
    cardinality :         624
    *height :             6-1
    * bats :              equal to handed
    * batting-average :   .252
    * team :
    * uniform-color :
Fielder
    isa :                 ML-Baseball-Player
    cardinality :         376
    *batting-average :    .262
Pee-Wee-Reese
    instance :            Fielder
    height :              5-10
    bats :                Right
    batting-average :     .309
    team :                Brooklyn-Dodgers
    uniform-color :       Blue
ML-Baseball-Team
    isa:                  Team
    cardinality :         26
    * team-size :         24
    * manager :
Brooklyn-Dodgers
    instance :            ML-Baseball-Team
    team-size :           24
    manager :             Leo-Durocher
    players :             {Pee-Wee-Reese,...}
```

Fig. 9.5 *A Simplified Frame System*

that it has a manager, and so on. The easiest way to allow for this is to go back to the idea of the Dodgers as an instance of *ML-Baseball-Team,* with the set of players given as a slot value.

But what we have encountered here is an example of a more general problem. A class is a set, and we want to be able to talk about properties that its elements possess. We want to use inheritance to infer those properties from general knowledge about the set. But a class is also an entity in itself. It may possess properties that belong not to the individual instances but rather to the class as a whole. In the case of *Brooklyn-Dodgers*, such properties included team size and the existence of a manager. We may even want to inherit some of these properties from a more general kind of set. For example, the Dodgers can inherit a default team size from the set of all major league baseball teams. To support this, we need to view a class as two things simultaneously: a subset (*isa*) of a larger class that also contains its elements and an instance (*instance*) of a class of sets, from which it inherits its set-level properties.

To make this distinction clear, it is useful to distinguish between regular classes, whose elements are individual entities, and *metaclasses*, which are special classes whose elements are themselves classes. A class is now an element of (*instance*) some class (or classes) as well as a subclass (*isa*) of one or more classes. A class inherits properties from the class of which it is an instance, just as any instance does. In addition, a class passes inheritable properties down from its superclasses to its instances.

Let us consider an example. Figure 9.6 shows how we could represent teams as classes using this distinction. Figure 9.7 shows a graphic view of the same classes. The most basic metaclass is the class *Class*. It represents the set of all classes. All classes are instances of it, either directly or through one of its subclasses. In the example, *Team* is a subclass (subset) of *Class* and *ML-Baseball-Team* is a subclass of *Team*. The class *Class* introduces the attribute *cardinality*, which is to be inherited by all instances of *Class* (including itself). This makes sense since all the instances of *Class* are sets and all sets have a cardinality.

Class		
	instance :	*Class*
	isa :	*Class*
	* *cardinality* :	
Team		
	instance :	*Class*
	isa :	*Class*
	cardinality :	{the number of teams that exist}
	**team-size* :	{each team has a size}
ML-Baseball-Team		
	isa :	*Mammal*
	instance :	*Class*
	isa :	*Team*
	cardi nality :	26 {the number of baseball teams that exist}
	* *team-size* :	24 {default 24 players on a team}
	* *manager* :	
Brooklyn-Dodgers		
	instance :	*ML-Baseball-Team*
	isa :	*ML-Baseball-Player*
	team-size :	24
	manager :	*Leo-Durocher*
	* *uniform-color* :	*Blue*
Pee-Wee-Reese		
	instance :	*Brooklyn-Dodgers*
	instance :	*Fielder*
	uniform-color :	*Blue*
	batting-average :	.309

Fig. 9.6 *Representing the Class of All Teams as a Metaclass*

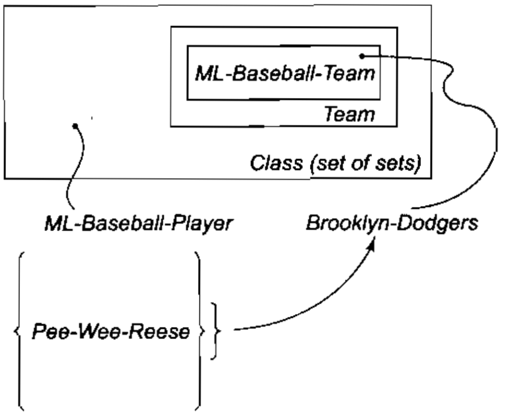


Fig. 9.7 *Classes and Metaclasses*

Team represents a subset of the set of all sets, namely those whose elements are sets of players on a team. It inherits the property of having a cardinality from *Class. Team* introduces the attribute *team-size,* which all its elements possess. Notice that *team-size* is like *cardinality* in that it measures the size of a set. But it applies to something different; *cardinality* applies to sets of sets and is inherited by all elements of *Class.* The slot *team-size* applies to the elements of those sets that happen to be teams. Those elements are sets of individuals.

ML-Baseball-Team is also an instance of *Class,* since it is a set. It inherits the property of having a cardinality from the set of which it is an instance, namely *Class.* But it is a subset of *Team.* All of its instances will have the property of having a *team-size* since they are also instances of the superclass *Team.* We have added at this level the additional fact that the default team size is 24, so all instances of *ML-Baseball-Team* will inherit that as well. In addition, we have added the inheritable slot *manager.*

Brooklyn-Dodgers is an instance of a *ML-Baseball-Team.* It is not an instance of *Class* because its elements are individuals, not sets. *Brooklyn-Dodgers* is a subclass of *ML-Baseball-Player* since all of its elements are also elements of that set. Since it is an instance of a *ML-Baseball-Team,* it inherits the properties *team-size* and *manager,* as well as their default values. It specifies a new attribute *uniform-color,* which is to be inherited by all of its instances (who will be individual players).

Finally, *Pee-Wee-Reese* is an instance of *Brooklyn-Dodgers.* That makes him also, by transitivity up *isa* links, an instance of *ML-Baseball-Player.* But recall that in our earlier example we also used the class *Fielder,* to which we attached the fact that fielders have above-average batting averages. To allow that here, we simply make Pee Wee an instance of *Fielder* as well. He will thus inherit properties from both *Brooklyn-Dodgers* and from *Fielder,* as well as from the classes above these. We need to guarantee that when multiple inheritance occurs, as it does here, that it works correctly. Specifically, in this case, we need to assure that *batting-average* gets inherited from *Fielder* and not from *ML-Baseball-Player* through *Brooklyn-Dodgers.* We return to this issue in Section 9.2.5.

In all the frame systems we illustrate, all classes are instances of the metaclass *Class.* As a result, they all have the attribute *cardinality.* We leave the class *Class,* the *isa* links to it, and the attribute *cardinality* out of our descriptions of our examples, though, unless there is some particular reason to include them.

Every class is a set. But not every set should be described as a class. A class describes a set of entities that share significant properties. In particular, the default information associated with a class can be used as a basis for inferring values for the properties of its individual elements. So there is an advantage to representing as a class those sets for which membership serves as a basis for nonmonotonic inheritance. Typically, these are sets in which membership is not highly ephemeral. Instead, membership is based on some fundamental structural or functional properties. To see the difference, consider the following sets:

- People
- People who are major league baseball players
- People who are on my plane to New York

The first two sets can be advantageously represented as classes, with which a substantial number of inheritable attributes can be associated. The last, though, is different. The only properties that all the elements of that set probably share are the definition of the set itself and some other properties that follow from the definition (e.g., they are being transported from one place to another). A simple set, with some associated assertions, is adequate to represent these facts; nonmonotonic inheritance is not necessary.

9.2.2 Other Ways of Relating Classes to Each Other

We have talked up to this point about two ways in which classes (sets) can be related to each other. $Class_1$ can be a subset of $Class_2$. Or, if $Class_2$ is a metaclass, then $Class_1$ can be an instance of $Class_2$. But there are other ways that classes can be related to each other, corresponding to ways that sets of objects in the world can be related.

One such relationship is *mutually-disjoint-with,* which relates a class to one or more other classes that are guaranteed to have no elements in common with it. Another important relationship is *is-covered-by* which relates a class to a set of subclasses, the union of which is equal to it. If a class *is-covered-by* a set S of mutually disjoint classes, then S is called *a partition* of the class.

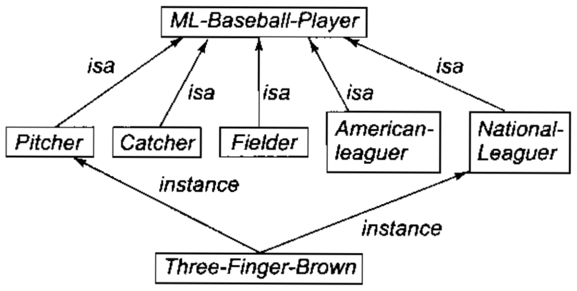
For examples of these relationships, consider the classes shown in Fig. 9.8, which represent two orthogonal ways of decomposing the class of major league baseball players. Everyone is either a pitcher, a catcher, or a fielder (and no one is more than one of these). In addition, everyone plays in either the National League or the American League, but not both.

9.2.3 Slots as Full-Fledged Objects

So far, we have provided a way to describe sets of objects and individual objects, both in terms of attributes and values. Thus we have made extensive use of attributes, which we have represented as slots attached to frames. But it turns out that there are several reasons why we would like to be able to represent attributes explicitly and describe their properties. Some of the properties we would like to be able to represent and use in reasoning include:

- The classes to which the attribute can be attached, i.e. for what classes does it make sense? For example, weight makes sense for physical objects but not for conceptual ones (except in some metaphorical sense).
- Constraints on either the type or the value of the attribute. For example, the age of a person must be a numeric quantity measured in some time frame, and it must be less than the ages of the person's biological parents.
- A value that all instances of a class must have by the definition of the class.
- A default value for the attribute.
- Rules for inheriting values for the attribute. The usual rule is to inherit down *isa* and *instance* links. But some attributes inherit in other ways. For example, *last-name* inherits down the *child-of* link.
- Rules for computing a value separately from inheritance. One extreme form of such a rule is a procedure written in some procedural programming language such as LISP.
- An inverse attribute.
- Whether the slot is single-valued or multivalued.

In order to be able to represent these attributes of attributes, we need to describe attributes (slots) as frames. These frames will be organized into an *isa* hierarchy, just as any other frames are, and that hierarchy can then be used to support inheritance of values for attributes of slots. Before we can describe such a hierarchy in detail, we need to formalize our notion of a slot.



ML-Baseball-Player	
is-covered-by :	*{Pitcher, Catcher, Fielder}*
	{American-Leaguer, National-Leaguer}
Pitcher	
isa :	*ML-Baseball-Player*
mutually-disjoint-with :	*{Catcher, Fielder}*
Catcher	
isa :	*ML-Baseball-Player*
mutually-disjoint-with:	*{Pitcher, Fielder}*
Fielder	
isa :	*ML-Baseball-Player*
mutually-disjoint-with :	*{Pitcher, Catcher}*
American-Leaguer	
isa :	*ML-Baseball-Player*
mutually-disjoint-with :	*{National-Leaguer}*
National-Leaguer	
isa :	*ML-Baseball-Player*
mutually-disjoint-with :	*{American-Leaguer}*
Three-Finger-Brown	
instance :	*Pitcher*
instance :	*National-Leaguer*

Fig. 9.8 *Representing Relationships among Classes*

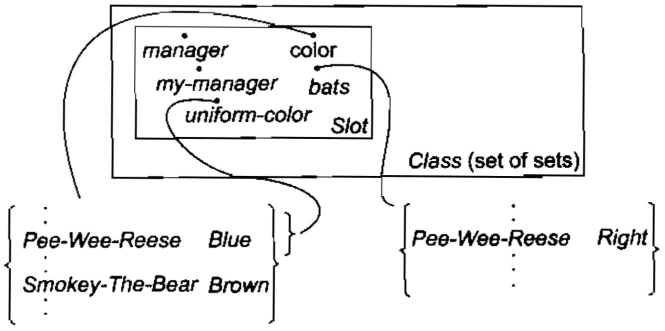
A slot is a relation. It maps from elements of its domain (the classes for which it makes sense) to elements of its range (its possible values). A relation is a set of ordered pairs. Thus it makes sense to say that one relation (R_1) is a subset of another (R_2). In that case, R_1 is a specialization of R_2, so in our terminology *isa* (R_1, R_2). Since a slot is a set, the set of all slots, which we will call *Slot*, is a metaclass. Its instances are slots, which may have subslots.

Figures 9.9 and 9.10 illustrate several examples of slots represented as frames. *Slot* is a metaclass. Its instances are slots (each of which is a set of ordered pairs). Associated with the metaclass are attributes that each instance (i.e., each actual slot) will inherit. Each slot, since it is a relation, has a domain and a range. We represent the domain in the slot labeled *domain*. We break up the representation of the range into two parts: *range* gives the class of which elements of the rangemust be elements; *range-constraint* contains a logical expression that further constrains the range to be elements of *range* that also satisfy the constraint. *If range-constraint* is absent, it is taken to be TRUE. The advantage to breaking the description apart into these two pieces is that type checking is much cheaper than is arbitrary constraint checking, so it is useful to be able to do it separately and early during some reasoning processes.

The other slots do what you would expect from their names. If there is a value for *definition*, it must be propagated to all instances of the slot. If there is a value for *default*, that value is inherited to all instances of the slot unless there is an overriding value. The attribute *transfers-through* lists other slots from which values for this slot can be derived through inheritance. The *to-compute* slot contains a procedure for deriving its value. The *inverse* attribute contains the inverse of the slot. Although in principle all slots have inverses, sometimes they are not useful enough in reasoning to be worth representing. And *single-valued* is used to mark the special cases in which the slot is a function and so can have only one value.

Of course, there is no advantage to representing these properties of slots if there is no reasoning mechanism that exploits them. In the rest of our discussion, we assume that the frame-system interpreter knows how to reason with all of these slots of slots as part of its built-in reasoning capability. In particular, we assume that it is capable of performing the following reasoning actions:

- Consistency checking to verify that when a slot value is added to a frame
  - The slot makes sense for the frame. This relies on the *domain* attribute of the slot.
  - The value is a legal value for the slot. This relies on the *range* and *range-constraints* attributes.
- Maintenance of consistency between the values for slots and their inverses when ever one is updated.
- Propagation of *definition* values along *isa* and *instance* links.
- Inheritance of *default* values along *isa* and *instance* links.



Slot	
isa :	*Class*
instance :	*Class*
* *domain* :	
* *range* :	
* *range-constraint* :	
* *definition* :	
* *default* :	
* *transfers-through* :	
* *to-compute* :	
* *inverse* :	
* *single-valued* :	

manager	
instance :	*Slot*
domain :	*ML-Baseball-Team*
range :	*Person*
range-constraint :	λx (*baseball-experience* x.*manager*)
default :	
inverse :	*manager-of*
single-valued :	TRUE

Fig. 9.9 *Representing Slots as Frames, I*

```
my-manager
    instance :            Slot
    domain :              ML-Baseball-Player
    range :               Person
    range-constraint :    λx (baseball-experience x.my-manager)
    to-compute :          λx (x.team).manager
    single-valued :       TRUE
color
    instance :            Slot
    domain :              Physical-Object
    range :               Color-Set
    transfers-through :   top-level-part-of
    visual-salience :     High
    single-valued :       FALSE
uniform-color
    instance :            Slot
    isa :                 color
    domain :              team-player
    range :               Color-Set
    range-constraint :    not Pink
    visual-salience :     High
    single-valued :       FALSE
bats
    instance :            Slot
    domain :              ML-Baseball-Player
    range :               {Left, Right, Switch)
    to-compute :          λx x.handed
    single-valued :       TRUE
```

Fig. 9.10 *Representing Slots as Frames, II*

- Computation of a value of a slot as needed. This relies on the *to-compute* and *transfers-through* attributes.
- Checking that only a single value is asserted *for single-valued* slots. This is usually done by replacing an old value by the new one when it is asserted. An alternative is to force explicit retraction of the old value and to signal a contradiction if a new value is asserted when another is already there.

There is something slightly counterintuitive about this way of defining slots. We have defined the properties *range-constraint* and *default* as parts of a slot. But we often think of them as being properties of a slot associated with a particular class. For example, in Fig. 9.5, we listed two defaults for the *batting-average* slot, one associated with major league baseball players and one associated with fielders. Figure 9.11 shows how

```
batting-average
    instance :            Slot
    domain :              ML-Baseball-Player
    range :               Number
    range-constraint :    λx (0 ≤ x.range-constraint ≤ 1)
    default :             .252
    single-valued :       TRUE
fielder-batting-average
    instance :            Slot
    isa :                 batting-average
    domain :              Fielder
    range :               Number
    range-constraint :    λx (0 ≤ x.range-constraint ≤ 1)
    default :             .262
    single-valued :       TRUE
```

Fig. 9.11 *Associating Defaults with Slots*

this can be represented correctly, by creating a specialization of *batting-average* that can be associated with a specialization of *ML-Baseball-Player* to represent the more specific information that is known about the specialized class. This seems cumbersome. It is natural, though, given our definition of a slot as a relation. There are really two relations here, one a specialization of the other. And below we will define inheritance so that it looks for values of either the slot it is given or any of that slot's generalizations.

Unfortunately, although this model of slots is simple and it is internally consistent, it is not easy to use. So we introduce some notational shorthand that allows the four most important properties of a slot (domain, range, definition, and default) to be defined implicitly by how the slot is used in the definitions of the classes in its domain. We describe the domain implicitly to be the class where the slot appears. We describe the range and any range constraints with the clause MUST BE, as the value of an inherited slot. Figure 9.12 shows an example of this notation. And we describe the definition and the default, if they are present, by inserting them as the value of the slot when it appears. The two will be distinguished by prefixing a definitional value with an asterisk (*). We then let the underlying bookkeeping of the frame system create the frames that represent slots as they are needed.

```
ML-Baseball-Player
    bats :                  MUST BE {Left, Right, Switch}
```

Fig. 9.12 *A Shorthand Notation for Slot-Range Specification*

Now let's look at examples of how these slots can be used. The slots *bats* and *my-manager* illustrate the use of the *to-compute* attribute of a slot. The variable *x* will be bound to the frame to which the slot is attached. We use the dot notation to specify the value of a slot of a frame. Specifically, *x.y* describes the value(s) of the *y* slot of frame *x*. So we know that to compute a frame's value for *my-manager,* it is necessary to find the frame's value for *team,* then find the resulting team's manager. We have simply composed two slots to form a new one.[3] Computing the value of the *bats* slot is even simpler. Just go get the value of the *handed* slot.

The *manager* slot illustrates the use of a range constraint. It is stated in terms of a variable *x,* which is bound to the frame whose *manager* slot is being described. It requires that any manager be not only a person but someone with baseball experience. It relies on the domain-specific function *baseball-experience,* which must be defined somewhere in the system.

The slots *color* and *uniform-color* illustrate the arrangement of slots in an *isa* hierarchy. The relation *color* is a fairly general one that holds between physical objects and colors. The attribute *uniform-color* is a restricted form of *color* that applies only between team players and the colors that are allowed for team uniforms (anything but pink). Arranging slots in a hierarchy is useful for the same reason that arranging any thing else in a hierarchy is: it supports inheritance. In this example, the general slot *color* is known to have high visual salience. The more specific slot *uniform-color* then inherits this property, so it too is known to have high visual salience.

The slot *color* also illustrates the use of the *transfers-through* slot, which defines a way of computing a slot's value by retrieving it from the same slot of a related object. In this example, we used *transfers-through* to capture the fact that if you take an object and chop it up into several top level parts (in other words, parts that are not contained inside each other), then they will all be the same color. For example, the arm of a sofa is the same color as the sofa. Formally, what *transfers-through* means in this example is

$$color(x, y) \wedge top\text{-}level\text{-}part\text{-}of(z, x) \rightarrow color(z, y)$$

In addition to these domain-independent slot attributes, slots may have domain-specific properties that support problem solving in a particular domain. Since these slots are not treated explicitly by the frame-system interpreter, they will be useful precisely to the extent that the domain problem solver exploits them.

[3]Notice that since slots are relations rather than functions, their composition may return a set of values.

9.2.4 Slot-Values as Objects

In the last section, we reified the notion of a slot by making it an explicit object that we could make assertions about. In some sense this was not necessary. A finite relation can be completely described by listing its elements. But in practical knowledge-based systems one often does not have that list. So it can be very important to be able to make assertions about the list without knowing all of its elements. Reification gave us a way to do this.

The next step along this path is to do the same thing to a particular attribute-value (an instance of a relation) that we did to the relation itself. We can reify it and make it an object about which assertions can be made. To see why we might want to do this, let us return to the example of John and Bill's height that we discussed in Section 9.1.3. Figure 9.13 shows a frame-based representation of some of the facts. We could easily record Bill's height if we knew it. Suppose, though, that we do not know it. All we know is that John is taller than Bill. We need a way to make an assertion about the value of a slot without knowing what that value is. To do that, we need to view the slot and its value as an object.

```
John
   height:        72
Bill
   height:
```

Fig. 9.13 *Representing Slot-Values*

We could attempt to do this the same way we made slots themselves into objects, namely by representing them explicitly as frames. There seems little advantage to doing that in this case, though, because the main advantage of frames does not apply to slot values: frames are organized into an *isa* hierarchy and thus support inheritance. There is no basis for such an organization of slot values. So instead, we augment our value representation language to allow the value of a slot to be stated as either or both of:

- A value of the type required by the slot.
- A logical constraint on the value. This constraint may relate the slot's value to he values of other slots or to domain constants.

If we do this to the frames of Fig. 9.13, then we get the frames of Fig. 9.14. We again use the lambda notation as a way to pick up the name of the frame that is being described.

```
John
   height:        72; λx (x.height > Bill.height)
Bill
   height:        λx (x.height < John.height)
```

Fig. 9.14 *Representing Slot-Values with Lambda Notation*

9.2.5 Inheritance Revisited

In Chapter 4, we presented a simple algorithm for inheritance. But that algorithm assumed that the *isa* hierarchy was a tree. This is often not the case. To support flexible representations of knowledge about the world, it is necessary to allow the hierarchy to be an arbitrary directed acyclic graph (DAG). We know that acyclic graphs are adequate because *isa* corresponds to the subset relation. Hierarchies that are not trees are called *tangled hierarchies.* Tangled hierarchies require a new inheritance algorithm. In the rest of this section, we discuss an algorithm for inheriting values for single-valued slots in a tangled hierarchy. We leave the problem of inheriting multivalued slots as an exercise.

Consider the two examples shown in Fig. 9.15 (in which we return to a network notation to make it easy to visualize the *isa* structure). In Fig. 9.15(a), we want to decide whether *Fifi* can fly. The correct answer is no.

Although birds in general can fly, the subset of birds, ostriches, does not. Although the class *Pet-Bird* provides a path from *Fifi* to *Bird* and thus to the answer that *Fifi* can fly, it provides no information that conflicts with the special case knowledge associated with the class *Ostrich*, so it should have no affect on the answer. To handle this case correctly, we need an algorithm for traversing the *isa* hierarchy that guarantees that specific knowledge will always dominate more general facts.

In Fig. 9.15(b), we return to a problem we discussed in Section 7.2.1, namely determining whether Dick is a pacifist. Again, we must traverse multiple *instance* links, and more than one answer can be found along the paths. But in this case, there is no well-founded basis for choosing one answer over the other. The classes that are associated with the candidate answers are incommensurate with each other in the partial ordering that is defined by the DAG formed by the *isa* hierarchy. Just as we found that in Default Logic this theory had two extensions and there was no principled basis for choosing between them, what we need here is an inheritance algorithm that reports the ambiguity; we do not want an algorithm that finds one answer (arbitrarily) and stops without noticing the other.

One possible basis for a new inheritance algorithm is path length. This can be implemented by executing a breadth-first search, starting with the frame for which a slot value is needed. Follow its *instance* links, then follow *isa* links upward. If a path produces a value, it can be terminated, as can all other paths once their length exceeds that of the successful path. This algorithm works for both of the examples in Fig. 9.15. In (a), it finds a value at *Ostrich*. It continues the other path to the same length (*Pet-Bird*), fails to find any other answers, and then halts. In the case of (b), it finds two competing answers at the same level, so it can report the contradiction.

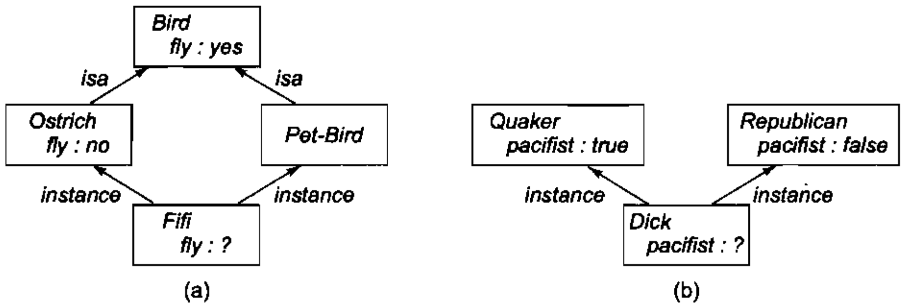


Fig. 9.15 *Tangled Hierarchies*

But now consider the examples shown in Fig. 9.16. In the case of (a), our new algorithm reaches *Bird* (via *Pet-Bird*) before it reaches *Ostrich*. So it reports that *Fifi* can fly. In the case of (b), the algorithm reaches *Quaker* and stops without noticing a contradiction. The problem is that path length does not always correspond to the level of generality of a class. Sometimes what it really corresponds to is the degree of elaboration of classes in the knowledge base. If some regions of the knowledge base have been elaborated more fully than others, then their paths will tend to be longer. But this should not influence the result of inheritance if no new information about the desired attribute has been added.

The solution to this problem is to base our inheritance algorithm not on path length but on the notion of *inferential distance* [Touretzky, 1986], which can be defined as follows:

$Class_1$ is closer to $Class_2$ than to $Class_3$, if and only if $Class_1$ has an inference path through $Class_2$ to $Class_3$ (in other words, $Class_2$ is between $Class_1$ and $Class_3$).

Notice that inferential distance defines only a partial ordering. Some classes are incommensurate with each other under it.

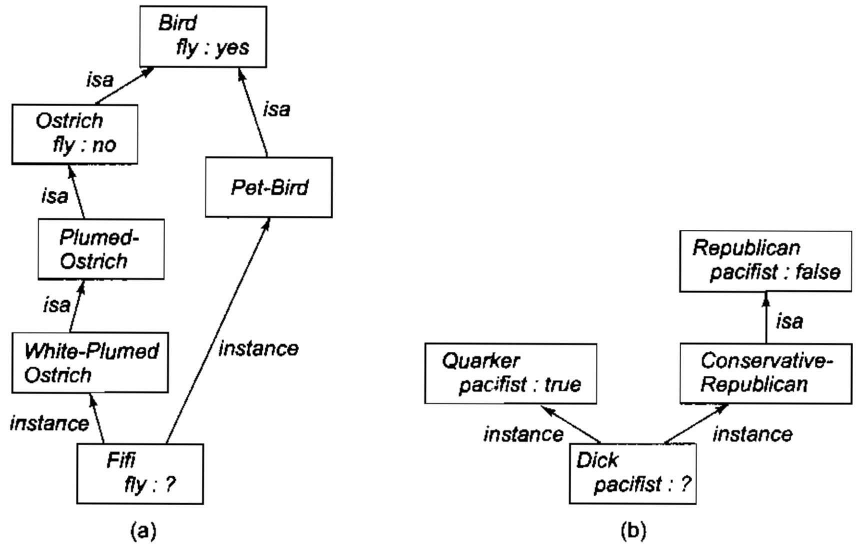


Fig. 9.16 *More Tangled Hierarchies*

We can now define the result of inheritance as follows: The set of competing values for a slot *S* in a frame *F* contains all those values that

- Can be derived from some frame *X* that is above *F* in the *isa* hierarchy
- Are not contradicted by some frame *Y* that has a shorter inferential distance to *F* than *X* does

Notice that under this definition competing values that are derived from incommensurate frames continue to compete.

Using this definition, let us return to our examples. For Fig. 9.15(a), we had two candidate classes from which to get an answer. But *Ostrich* has a shorter inferential distance to *Fifi* than *Bird* does, so we get the single answer no. For Fig. 9.15(b), we get two answers, and neither is closer to *Dick* than the other, so we correctly identify a contradiction. For Fig. 9.16(a), we get two answers, but again *Ostrich* has a shorter inferential distance to *Fifi* than *Bird* does. The significant thing about the way we have defined inferential distance is that as long as *Ostrich* is a subclass of *Bird,* it will be closer to all its instances than *Bird* is, no matter how many other classes are added to the system. For Fig. 9.16(b), we again get two answers and again neither is closer to *Dick* than the other.

There are several ways that this definition can be implemented as an inheritance algorithm. We present a simple one. It can be made more efficient by caching paths in the hierarchy, but we do not do that here.

Algorithm: Property Inheritance

To retrieve a value *V* for slot S of an instance *F* do:

1. Set *CANDIDATES* to empty.
2. Do breadth-first or depth-first search up the *isa* hierarchy from *F,* following all *instance* and *isa* links. At each step, see if a value for *S* or one f its generalizations is stored.
   (a) If a value is found, add it to *CANDIDATES* and terminate that branch of the search.
   (b) If no value is found but there are *instance* or *isa* links upward, follow them.
   (c) Otherwise, terminate the branch.

3. For each element C of *CANDIDATES* do:
   (a) See if there is any other element of *CANDIDATES* that was derived from a class closer to *F* than the class from which *C* came.
   (b) If there is, then, remove *C* from *CANDIDATES.*
4. Check the cardinality of *CANDIDATES:*
   (a) If it is 0, then report that no value was found.
   (b) If it is 1, then return the single element of *CANDIDATES* as *V*.
   (c) If it is greater than 1, report a contradiction.

This algorithm is guaranteed to terminate because the *isa* hierarchy is represented as an acyclic graph.

9.2.6 Frame Languages

The idea of a frame system as a way to represent declarative knowledge has been encapsulated in a series of frame-oriented knowledge representation languages, whose features have evolved and been driven by an increased understanding of the sort of representation issues we have been discussing. Examples of such languages include KRL [Bobrow and Winograd, 1977], FRL [Roberts and Goldstein, 1977], RLL [Greiner and Lenat, 1980], KL-ONE [Brachman, 1979; Brachman and Schmolze, 1985], KRYPTON [Brachman *et al.*, 1985], NIKL [Kaczmarek *et al.*, 1986], CYCL [Lenat and Guha, 1990], conceptual graphs [Sowa, 1984], THEO [Mitchell *et al.*, 1989], and FRAMEKIT [Nyherg, 1988]. Although not all of these systems support all of the capabilities that we have discussed, the more modern of these systems permit elaborate and efficient representation of many kinds of knowledge. Their reasoning methods include most of the ones described here, plus many more, including subsumption checking, automatic classification, and various methods for consistency maintenance.

EXERCISES

1. Construct semantic net representations for the following:
   (a) *Pompeian(Marcus), Blacksmith(Marcus)*
   (b) Mary gave the green flowered vase to her favorite cousin.
2. Suppose we want to use a semantic net to discover relationships that could help in disambiguating the word "bank" in the sentence

   John went downtown to deposit his money in the bank.

   The financial institution meaning for bank should be preferred over the river bank meaning.
   (a) Construct a semantie net that contains representations for the relevant concepts.
   (b) Show how intersection seareh could be used to find the connection between the correct meaning for bank and the rest of the sentence more easily than it can find a connection with the incorrect meaning.
3. Construct partitioned semantic net representations for the following:
   (a) Every batter hit a ball.
   (b) All the batters like the pitcher.
4. Construct one consistent frame representation of all the baseball knowledge that was described in this chapter. You will need to choose between the two representations for team that we considered.
5. Modify the property inheritance algorithm of Section 9.2 to work for multiple-valued attributes, sueh as the attribute *believes-in-principles*, defined as follows:

believes-in-principles	
instance :	*Slot*
domain :	*Person*
range :	*Philosophical-Principles*
single-valued :	FALSE

6. Define the value of a multiple-valued slot *S* of class *C* to be the union of the values that are found for *S* and all its generalizations at *C* and all its generalizations. Modify your technique to allow a class to exclude specific values that are associated with one or more of its superclasses.
7. Pick a problem area and represent some knowledge about it the way we represented baseball knowledge in this chapter.
8. How would you classify and represent the various types of triangles?

CHAPTER

10

STRONG SLOT-AND-FILLER STRUCTURES

In the 1960s and 1970s, students frequently asked, "Which kind of representation is best?" and I usually replied that we'd need more research... But now I would reply: To solve really hard problems, we'll have to use several different representations. This is because each particular kind of data structure has its own virtues and deficiencies, and none by itself would seem adequate for all the different functions involved with what we call common sense.

—Minsky, Marvin
(1927-), American cognitive scientist

The slot-and-filler structures described in the previous chapter are very general, Individual semantic networks and frame systems may have specialized links and inference procedures, but there are no hard and fast rules about what kinds of objects and links are good in general for knowledge representation. Such decisions are left up to the builder of the semantic network or frame system.

The three structures discussed in this chapter, *conceptual dependency, scripts,* and *CYC,* on the other hand, embody specific notions of what types of objects and relations are permitted. They stand for powerful theories of how AI programs can represent and use knowledge about common situations.

10.1 CONCEPTUAL DEPENDENCY

Conceptual dependency (often nicknamed CD) is a theory of how to represent the kind of knowledge about events that is usually contained in natural language sentences. The goal is to represent the knowledge in a way that

- Facilitates drawing inferences from the sentences.
- Is independent of the language in which the sentences were originally stated.

Because of the two concerns just mentioned, the CD representation of a sentence is built not out of primitives corresponding to the words used in the sentence, but rather out of conceptual primitives that can be combined to form the meanings of words in any particular language. The theory was first described in Schank [1973] and was further developed in Schank [1975]. It has since been implemented in a variety of programs that read and understand natural language text. Unlike semantic nets, which provide only a structure into which nodes representing information at any level can be placed, conceptual dependency provides both a structure and a specific set of primitives, at a particular level of granularity, out of which representations of particular pieces of information can be constructed.

$$\begin{tikzcd} & & & & \text{man} \\ I \arrow[r, Leftrightarrow, "p"] & \text{ATRANS} & \text{book} \arrow[l, "o"'] & \text{R} \arrow[l] \arrow[ur, "to"] & \\ & & & & I \arrow[ul, "from"] \end{tikzcd}$$

where the symbols have the following meanings:

- Arrows indicate direction of dependency.
- Double arrow indicates two way link between actor and action.
- p indicates past tense.
- ATRANS is one of the primitive acts used by the theory. It indicates transfer of possession.
- o indicates the object case relation.
- R indicates the recipient case relation.

Fig. 10.1 *A Simple Conceptual Dependency Representation*

As a simple example of the way knowledge is represented in CD, the event represented by the sentence

I gave the man a book.

would be represented as shown in Fig. 10.1.

In CD, representations of actions are built from a set of primitive acts. Although there are slight differences in the exact set of primitive actions provided in the various sources on CD, a typical set is the following, taken from Schank and Abelson [1977]:

ATRANS	Transfer of an abstract relationship (e.g., give)
PTRANS	Transfer of the physical location of an object (e.g., go)
PROPEL	Application of physical force to an object (e.g., push)
MOVE	Movement of a body part by its owner (e.g., kick)
GRASP	Grasping of an object by an actor (e.g., clutch)
INGEST	Ingestion of an object by an animal (e.g., eat)
EXPEL	Expulsion of something from the body of an animal (e.g., cry)
MTRANS	Transfer of mental information (e.g., tell)
MBUILD	Building new information out of old (e.g., decide)
SPEAK	Production of sounds (e.g., say)
ATTEND	Focusing of a sense organ toward a. stimulus (e.g., listen)

A second set of CD building blocks is the set of allowable dependencies among the conceptualizations described in a sentence. There are four primitive conceptual categories from which dependency structures can be built. These are

ACTs	Actions
PPs	Objects (picture producers)
AAs	Modifiers of actions (action aiders)
PAs	Modifiers of PPs (picture aiders)

In addition, dependency structures are themselves conceptualizations and can serve as components of larger dependency structures.

The dependencies among conceptualizations correspond to semantic relations among the underlying concepts. Figure 10.2 lists the most important ones allowed by CD.[1] The first column contains the rules; the second contains examples of their-useLand the third contains an English version of each example. The rules shown in the Fig. can be interpreted as follows:

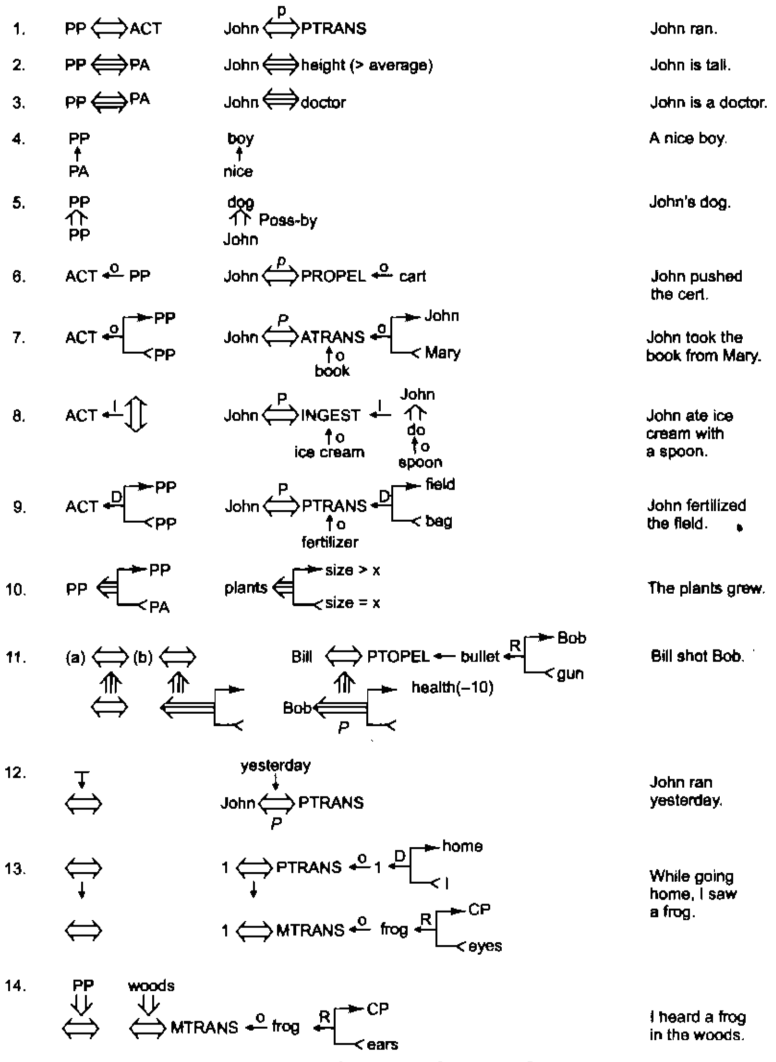

1. PP ⟺ ACT — John ⟺(p) PTRANS — John ran.
2. PP ⟺ PA — John ⟺ height (> average) — John is tall.
3. PP ⟺ PA — John ⟺ doctor — John is a doctor.
4. PP ↑ PA — boy ↑ nice — A nice boy.
5. PP ⇑ PP — dog ⇑ Poss-by John — John's dog.
6. ACT ←o PP — John ⟺(p) PROPEL ←o cart — John pushed the cart.
7. ACT ←o PP / PP — John ⟺(p) ATRANS ←o John / Mary, ↑o book — John took the book from Mary.
8. ACT ←I ⇕ — John ⟺(p) INGEST ←I John ⇑ do ↑o spoon, ↑o ice cream — John ate ice cream with a spoon.
9. ACT ←D PP / PP — John ⟺(p) PTRANS ←D field / bag, ↑o fertilizer — John fertilized the field.
10. PP ⟸ PP / PA — plants ⟸ size > x / size = x — The plants grew.
11. (a) ⟺ (b) ⟺ — Bill ⟺ PTOPEL ← bullet ←R Bob / gun; Bob ⟸(p) health(−10) — Bill shot Bob.
12. T ⟺ — yesterday ↓ John ⟺(p) PTRANS — John ran yesterday.
13. ⟺ ↓ ⟺ — 1 ⟺ PTRANS ←o 1 ←D home / I; 1 ⟺ MTRANS ←o frog ←R CP / eyes — While going home, I saw a frog.
14. PP ⇓ ⟺ — woods ⇓ ⟺ MTRANS ←o frog ←R CP / ears — I heard a frog in the woods.

Fig. 10.2 *The Dependencies of CD*

- Rule 1 describes the relationship between an actor and the event he or she causes. This is a two-way dependency since neither actor nor event can be considered primary. The letter p above the dependency link indicates past tense.
- Rule 2 describes the relationship between a PP and a PA that is being asserted to describe it. Many state descriptions, such as height, are represented in CD as numeric scales.

[1]The lable shown in the figure is adapted from several tables in Schank [1973].

- Rule 3 describes the relationship between two PPs, one of which belongs to the set defined by the other.
- Rule 4 describes the relationship between a PP and an attribute that has already been predicated of it. The direction of the arrow is toward the PP being described.
- Rule 5 describes the relationship between two PPs, one of which provides a particular kind of information about the other. The three most common types of information to be provided in this way are possession (shown as POSS-BY), location (shown as LOC), and physical containment (shown as CONT). The direction of the arrow is again toward the concept being described.
- Rule 6 describes the relationship between an ACT and the PP that is the object of that ACT. The direction of the arrow is toward the ACT since the context of the specific ACT determines the meaning of the object relation.
- Rule 7 describes the relationship between an ACT and the source and,the recipient of the ACT.
- Rule 8 describes the relationship between an ACT and the instrument with which it is performed. The instrument must always be a full conceptualization (i.e.. it must contain an ACT), not just a single physical object.
- Rule 9 describes the relationship between an ACT and its physical source and destination.
- Rule 10 represents the relationship between a PP and a state in which it started and another in which it ended.
- Rule 11 describes the relationship between one conceptualization and another that causes it. Notice that the arrows indicate dependency of one conceptualization on another and so point in the opposite direction of the implication arrows. The two forms of the rule describe the cause of an action and the cause of a state change.
- Rule 12 describes the relationship between a conceptualization and the time at which the event it describes occurred.
- Rule 13 describes the relationship between one conceptualization and another that is the time of the first. The example for this rule also shows how CD exploits a model of the human information processing system; *see* is represented as the transfer of information between the eyes and the conscious processor.
- Rule 14 describes the relationship between a conceptualization and the place at which it occurred.

Conceptualizations representing events can be modified in a variety of ways to supply information normally indicated in language by the tense, mood, or aspect of a verb form. The use of the modifier p to indicate past tense has already been shown. The set of conceptual tenses proposed by Schank [1973] includes

p	Past
f	Future
t	Transition
t_s	Start transition
t_f	Finished transition
k	Continuing
?	Interrogative
/	Negative
nil	Present
delta	Timeless
c	Conditional

As an example of the use of these tenses, consider the CD representation shown in Fig. 10.3 (taken from Schank [1973]) of the sentence

Since smoking can kill you, I stopped.

The vertical causality link indicates that smoking kills one. Since it is marked c, however, we know only that smoking can kill one, not that it necessarily does. The horizontal causality link indicates that it is that first causality that made me stop smoking. The qualification t_{fp} attached to the dependency between I and INGEST indicates that the smoking (an instance of INGESTING) has stopped and that the stopping happened in the past.

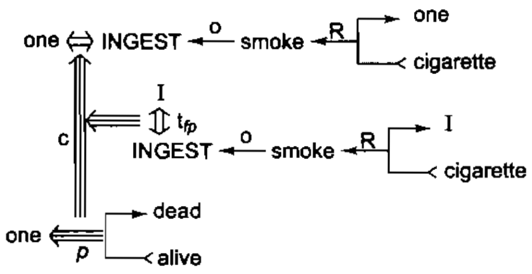


Fig. 10.3 *Using Conceptual Tenses*

There are three important ways in which representing knowledge using the conceptual dependency model facilitates reasoning with the knowledge:

1. Fewer inference rules are needed than would be required if knowledge were not broken down into primitives.
2. Many inferences are already contained in the representation itself.
3. The initial structure that is built to represent the information contained in one sentence will have holes that need to be filled. These holes can serve as an attention focuser for the program that must understand ensuing sentences.

Each of these points merits further discussion.

The first argument in favor of representing knowledge in terms of CD primitives rather than in the higher-level terms in which it is normally described is that using the primitives makes it easier to describe the inference rules by which the knowledge can be manipulated. Rules need only be represented once for each primitive ACT rather than once for every word that describes that ACT. For example, all of the following verbs involve a transfer of ownership of an object:

- Give
- Steal
- Take
- Donate

If any of them occurs, then inferences about who now has the object and who once had the object (and thus who may know something about it) may be important. In a CD representation, those possible inferences can be stated once and associated with the primitive ACT ATRANS.

A second argument in favor of the use of CD representation is that to construct it, we must use not only the information that is stated explicitly in a sentence but also a set of inference rules associated with the specific information. Having applied these: rules once, we store these results as part of the representation and they can be used repeatedly without the rules being reapplied. For example, consider the sentence

Bill threatened John With a broken nose.

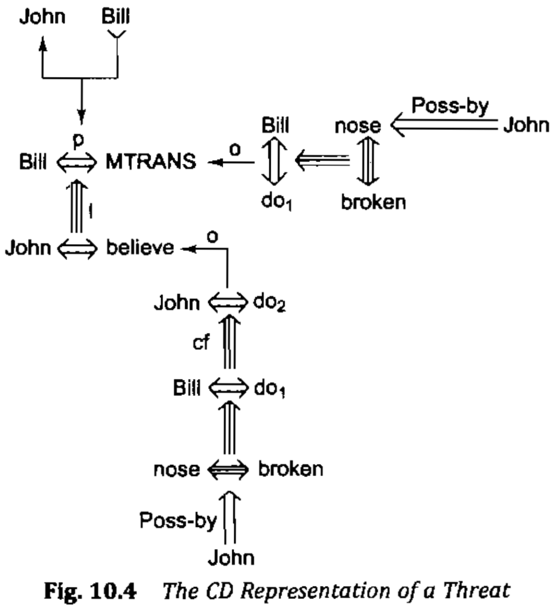


Fig. 10.4 *The CD Representation of a Threat*

The CD representation of the information contained in this sentence is shown in Fig. 10.4. (For simplicity, *believe* is shown as a single unit. In fact, it must be represented in terms of primitive ACTs and a model of the human information processing system.) It says that Bill informed John that he (Bill) will do something to break John's nose. Bill did this so that John will believe that if he (John) does some other thing (different from what Bill will do to break his nose), then Bill will break John's nose. In this representation, the word "believe" has been used to simplify the example. But the idea behind *believe* can be represented in CD as an MTRANS of a fact into John's memory. The actions do_1 and do_2 are dummy placeholders that refer to some as yet unspecified actions.

A third argument for the use of the CD representation is that unspecified elements of the representation of one piece of, information can be used as a focus for the understanding of later events as they are encountered. So, for example, after hearing that

Bill threatened John with a broken nose.

we might expect to find out what action Bill was trying to prevent John from performing. That action could then be substituted for the dummy action represented in Fig. 10.4 as do_2. The presence of such dummy objects provides clues as to what other events or objects are important for the understanding of the known event.

Of course, there are also arguments against the use of CD as a representation formalism. For one thing, it requires that all knowledge be decomposed into fairly low-level primitives. In Section 4.3.3 we discussed how this may be inefficient or perhaps even impossible in some situations. As Schank and Owens [1987] put it,

CD is a theory of representing fairly simple actions. To express, for example, "John bet Sam fifty dollars that the Mets would win the World Series" takes about two pages of CD forms. This does not seem reasonable.

Thus, although there are several arguments in favor of the use of CD as a model for representing events, it is not always completely appropriate to do so, and jt may be worthwhile to seek out higher-level primitives.

Another difficulty with the theory of conceptual dependency as a general model for the representation of knowledge is that it is only a theory of the representation of events. But to represent all the information that a complex program may need, it must be able to represent other things besides events. There have been attempts to define a set of primitives, similar to those of CD for actions, that can be used to describe other kinds of knowledge. For example, physical objects, which in CD are simply represented as atomic units, have been analyzed in Lehnert [1978]. A similar analysis of social actions is provided in Schank and Carbonell [1979]. These theories continue the style of representation pioneered by CD, but they have not yet been subjected to the same amount of empirical investigation (i.e,, use in real programs) as CD.

We have discussed the theory of conceptual dependency in some detail in order to illustrate the behavior of a knowledge representation system built around a fairly small set of specific primitive elements. But CD is not the only such theory to have been developed and used in AI programs. For another example of a primitive-based system, see Wilks [1972].

10.2 SCRIPTS

CD is a mechanism for representing and reasoning about events. But rarely do events occur in isolation. In this section, we present a mechanism for representing knowledge about common sequences of events.

A *script* is a structure that describes a stereotyped sequence of events in a particular context. A script consists of a set of slots. Associated with each slot may be some information about what kinds of values it may contain as well as a default value to be used if no other information is available. So far, this definition of a script looks very similar to that of a frame given in Section 9.2, and at this level of detail, the two structures are identical. But now, because of the specialized role to be played by a script, we can make some more precise statements about its structure.

Figure 10.5 shows part of a typical script, the restaurant script (taken from Schank and Abelson [1977]). It illustrates the important components of a script:

Entry conditions	Conditions that must, in general, be satisfied before the events described in the script can occur.
Result	Conditions that will, in general, be true after the events described in the script have occurred.
Props	Slots representing objects that are involved in the events described in the script. The presence of these objects can be inferred even if they are not mentioned explicitly.
Roles	Slots representing people who are involved in the events described in the script. The presence of these people, too, can be inferred even if they are not mentioned explicitly. If specific individuals are mentioned, they can be inserted into the appropriate slots.
Track	The specific variation on a more general pattern that is represented by this particular script. Different tracks of the same script will share many but not all components.
Scenes	The actual sequences of events that occur. The events are represented in conceptual dependency formalism.

Scripts are useful because, in the real world, there are patterns to the occurrence of events. These patterns arise because of causal relationships between events. Agents will perform one action so that they will then be able to perform another. The events described in a script form a giant *causal chain*. The beginning of the chain is the set of entry conditions which enable the first events of the script to occur. The end of the chain is the set of results which may enable later events or event sequences (possibly described by other scripts) to occur. Within the chain, events are connected both to earlier events that make them possible and to later events that they enable.

If a particular script is known to be appropriate in a given situation, then it can be very useful in predicting the occurrence of events that were not explicitly mentioned. Scripts can also be useful by indicating how events that were mentioned relate to each other. For example, what is the connection between someone's ordering steak and someone's eating steak? But before a particular script can be applied, it must be activated (i.e., it must be selected as appropriate to the current situation). There are two ways in which it may be useful to activate a script, depending on how important the script is likely to be:

- For fleeting scripts (ones that are mentioned briefly and may be referred to again but are not central to the situation), it may be sufficient merely to store a pointer to the script so that it can be accessed later if necessary. This would be an appropriate strategy to take with respect to the restaurant script when confronted with a story such as

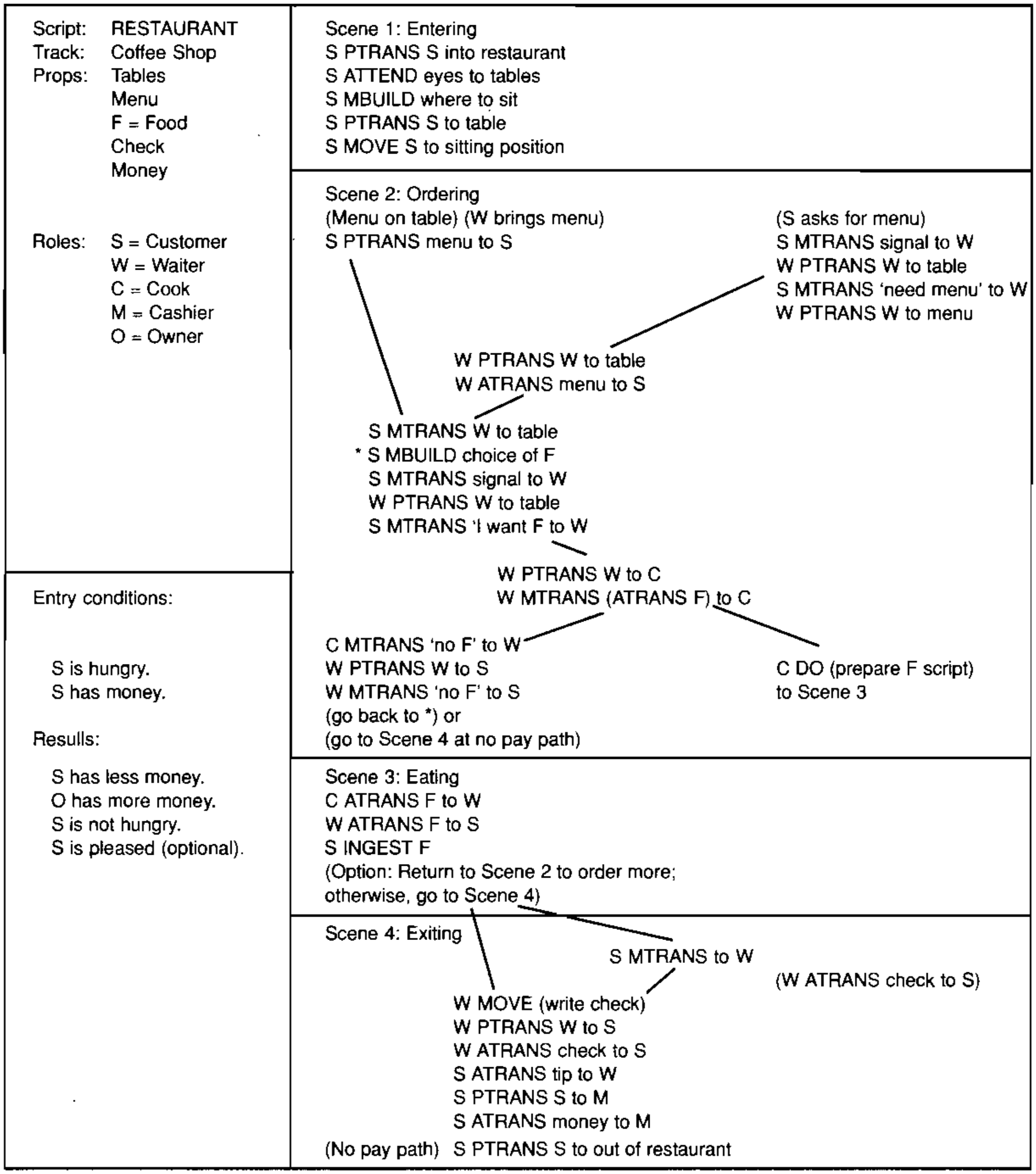
  Susan passed her favorite restaurant on her way to the museum. She really enjoyed the new Picasso exhibit.

- For nonfleeting scripts it is appropriate to activate the script fully and to attempt to fill in its slots with particular objects and people involved in the current situation.

The headers of a script (its preconditions, its preferred locations, its props, its roles, and its events) can all serve as indicators that the script should be activated. In order to cut down on the number of times a spurious script is activated, it has proved useful to require that a situation contain at least two of a script's headers before the script will be activated.

Once a script has been activated, there are, as we have already suggested, a variety of ways in which it can be useful in interpreting a particular situation. The most important of these is the ability to predict events that have not explicitly been observed. Suppose, for example, that you are told the following story:

John went out to a restaurant last night. He ordered steak. When he paid for it, he noticed that he was running out of money. He hurried home since it had started to rain.



Script: RESTAURANT
Track: Coffee Shop
Props: Tables
Menu
F = Food
Check
Money

Roles: S = Customer
W = Waiter
C = Cook
M = Cashier
O = Owner

Entry conditions:

S is hungry.
S has money.

Results:

S has less money.
O has more money.
S is not hungry.
S is pleased (optional).

Scene 1: Entering
S PTRANS S into restaurant
S ATTEND eyes to tables
S MBUILD where to sit
S PTRANS S to table
S MOVE S to sitting position

Scene 2: Ordering
(Menu on table) (W brings menu) (S asks for menu)
S PTRANS menu to S
S MTRANS signal to W
W PTRANS W to table
S MTRANS 'need menu' to W
W PTRANS W to menu

W PTRANS W to table
W ATRANS menu to S

S MTRANS W to table
* S MBUILD choice of F
S MTRANS signal to W
W PTRANS W to table
S MTRANS 'I want F to W

W PTRANS W to C
W MTRANS (ATRANS F) to C

C MTRANS 'no F' to W
W PTRANS W to S
W MTRANS 'no F' to S
(go back to *) or
(go to Scene 4 at no pay path)

C DO (prepare F script)
to Scene 3

Scene 3: Eating
C ATRANS F to W
W ATRANS F to S
S INGEST F
(Option: Return to Scene 2 to order more;
otherwise, go to Scene 4)

Scene 4: Exiting
S MTRANS to W
(W ATRANS check to S)
W MOVE (write check)
W PTRANS W to S
W ATRANS check to S
S ATRANS tip to W
S PTRANS S to M
S ATRANS money to M
(No pay path) S PTRANS S to out of restaurant

Fig. 10.5 *The Restaurant Script*

If you were then asked the question

> Did John eat dinner last night?

you would almost certainly respond that he did, even though you were not told so explicitly. By using the restaurant script, a computer question-answerer would also be able to infer that John ate dinner, since the restaurant script could have been activated. Since all of the events in the story correspond to the sequence of events predicted by the script, the program could infer that the entire sequence predicted by the script occurred normally. Thus it could conclude, in particular, that John ate. In their ability to predict unobserved events, scripts are similar to frames and to other knowledge structures that represent stereotyped situations. Once one of these structures is activated in a particular situation, many predictions can be made.

A second important use of scripts is to provide a way of building a single coherent interpretation from a collection of observations. Recall that a script can be viewed as a giant causal chain. Thus it provides information about how events are related to each other. Consider, for example, the following story:

> Susan went out to lunch. She sat down at a table and called the waitress. The waitress brought her a menu and she ordered a hamburger.

Now consider the question

> Why did the waitress bring Susan a menu?

The script provides two possible answers to that question:

- Because Susan asked her to. (This answer is gotten by going backward in the causal chain to find out what caused her to do it.)
- So that Susan could decide what she wanted to eat. (This answer is gotten by going forward in the causal chain to find out what event her action enables.)

A third way in which a script is useful is that it focuses attention on unusual events. Consider the following story:

> John went to a restaurant. He was shown to his table. He ordered a large steak. He sat there and waited for a long time. He got mad and left.

The important part of this story is the place in which it departs from the expected sequence of events in a restaurant. John did not get mad because he was shown to his table. He did get mad because he had to wait to be served. Once the typical sequence of events is interrupted, the script can no longer be used to predict other events. So, for example, in this story, we should not infer that John paid his bill. But we can infer that he saw a menu, since reading the menu would have occurred before the interruption. For a discussion of SAM, a program that uses scripts to perform this kind of reasoning, see Cullingford [1981].

From these examples, we can see how information about typical sequences of events, as represented in scripts, can be useful in interpreting a particular, observed sequence of events. The usefulness of a script in some of these examples, such as the one in which unobserved events were predicted, is similar to the usefulness of other knowledge structures, such as frames. In other examples, we have relied on specific properties of the information stored in a script, such as the causal chain represented by the events it contains. Thus although scripts are less general structures than are frames, and so are not suitable for representing all kinds of knowledge, they can be very effective for representing the specific kinds of knowledge for which they were designed.

significant. Samuel's functions included, in addition to the obvious one, piece advantage, such things as capability for advancement, control of the center, threat of a fork, and mobility. These factors were then combined by attaching to each an appropriate weight and then adding the terms together. Thus the complete evaluation function had the form:

$$c_1 \times pieceadvantage + c_2 \times advancement + c_3 \times centercontrol...$$

There were also some nonlinear terms reflecting combinations of these factors. But Samuel did not know the correct weights to assign to each of the components. So he employed a simple learning mechanism in which components that had suggested moves that turned out to lead to wins were given an increased weight, while the weights of those that had led to losses were decreased.

Unfortunately, deciding which moves have contributed to wins and which to losses is not always easy. Suppose we make a very bad move, but then, because the opponent makes a mistake, we ultimately win the game. We would not like to give credit for winning to our mistake. The problem of deciding which of a series of actions is actually responsible for a particular outcome is called the *credit assignment problem* [Minsky, 1963]. It plagues many learning mechanisms, not just those involving games. Despite this and other problems, though, Samuel's checkers program was eventually able to beat its creator. The techniques it used to acquire this performance are discussed in more detail in Chapter 17.

We have now discussed the two important knowledge-based components of a good game-playing program: a good plausible-move generator and a good static evaluation function. They must both incorporate a great deal of knowledge about the particular game being played. But unless these functions are perfect, we also need a search procedure that makes it possible to look ahead as many moves as possible to see what may occur. Of course, as in other problem-solving domains, the role of search can be altered considerably by altering the amount of knowledge that is available to it. But, so far at least, programs that play nontrivial games rely heavily on search.

What search strategy should we use then? For a simple one-person game or puzzle, the A* algorithm described in Chapter 3 can be used. It can be applied to reason forward from the current state as far as possible in the time allowed. The heuristic function h' can be applied at terminal nodes and used to propagate values back up the search graph so that the best next move can be chosen. But because of their adversarial nature, this procedure is inadequate for two-person games such as chess. As values are passed back up, different assumptions must be made at levels where the program chooses the move and at the alternating levels where the opponent chooses. There are several ways that this can be done. The most commonly used method is the *minimax* procedure, which is described in the next section. An alternative approach is the B* algorithm [Berliner, 1979a], which works on both standard problem-solving trees and on game trees.

12.2 THE MINIMAX SEARCH PROCEDURE

The *minimax search procedure'is* a depth-first, depth-limited search procedure. It was described briefly in Section 1.3.1. The idea is to start at the current position and use the plausible-move generator to generate the set of possible successor positions. Now we can apply the static evaluation function to those positions and simply choose the best one. After doing so, we can back that value up to the starting position to represent our evaluation of it. The starting position is exactly as good for us as the position generated by the best move we can make next. Here we assume that the static evaluation function returns large values to indicate good situations for us, so our goal is to *maximize* the value of the static evaluation function of the next board position.

An example of this operation is shown in Fig. 12.1. It assumes a static evaluation function that returns values ranging from – 10 to 10, with 10 indicating a win for us, – 10 a win for the opponent, and 0 an even match. Since our goal is to maximize the value of the heuristic function, we choose to move to B. Backing B's value up to A, we can conclude that A's value is 8, since we know we can move to a position with a value of 8.

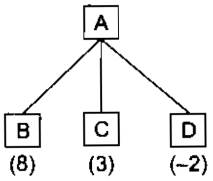


Fig. 12.1 *One-Ply Search*

But since we know that the static evaluation function is not completely accurate, we would like to carry the search farther ahead than one ply. This could be very important, for example, in a chess game in which we are in the middle of a piece exchange. After our move, the situation would appear to be very good, but, if we look one move ahead, we will see that one of our pieces also gets captured and so the situation is not as favorable as it seemed. So we would like to look ahead to see what will happen to each of the new game positions at-the next move which will be made by the opponent. Instead of applying the static evaluation function to each of the positions that we just generated, we apply the plausible-move generator, generating a set of successor positions for each position. If we wanted to stop here, at two-ply lookahead, we could apply the static evaluation function to each of these positions, as shown in Fig. 12.2.

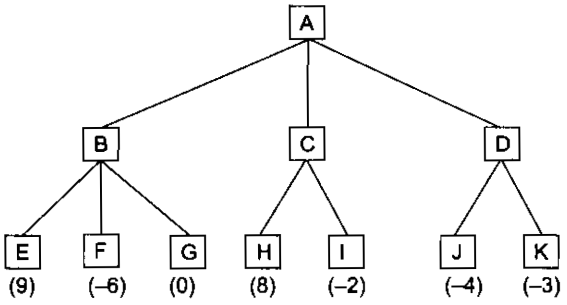


Fig. 12.2 *Two-Ply Search*

But now we must take into account that the opponent gets to choose which successor moves to make and thus which terminal value should be backed up to the next level. Suppose we made move B. Then the opponent must choose among moves E, F, and G. The opponent's goal is to *minimize* the value of the evaluation function, so he or she can be expected to choose move F. This means that if we make move B, the actual position in which we will end up one move later is very bad for us. This is true even though a possible configuration is that represented by node E, which is very good for us. But since at this level we are not the ones to move, we will not get to choose it. Figure 12.3 shows the result of propagating the new values up the tree. At the level representing the opponent's choice, the minimum value was chosen and backed up. At the level representing our choice, the maximum value was chosen.

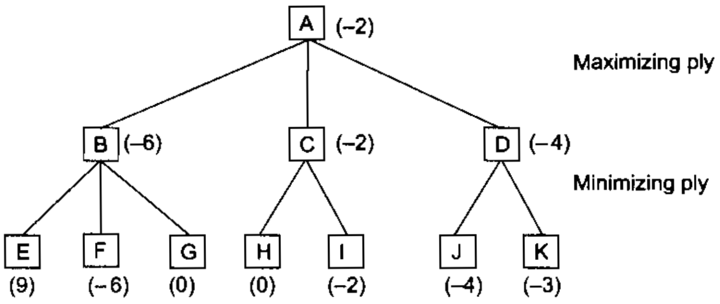


Fig. 12.3 *Backing Up the Values of a Two-Ply Search*

Once the values from the second ply are backed up, it becomes clear that the correct move for us to make at the first level, given the information we have available, is C, since there is nothing the opponent can do from there to produce a value worse than –2. This process can be repeated for as many ply as time allows, and the more accurate evaluations that are produced can be used to choose the correct move at the top level. The alternation of maximizing and minimizing at alternate ply when evaluations are being pushed back up corresponds to the opposing strategies of the two players and gives this method the name minimax.

Having described informally the operation of the minimax procedure, we now describe it precisely. It is a straightforward recursive procedure that relies on two auxiliary procedures that are specific to the game being played:

1. MOVEGEN(*Position, Player*)—The plausible-move generator, which returns a list of nodes representing the moves that can be made by *Player* in *Position.* We call the two players PLAYER-ONE and PLAYER-TWO; in a chess program, we might use the names BLACK and WHITE instead.
2. STATIC(*Position, Player*)—The static evaluation function, which returns a num ber representing the goodness of *Position* from the standpoint of *Player.*[2]

As with any recursive program, a critical issue in the design of the MINIMAX procedure is when to stop the recursion and simply call the static evaluation function. There are a variety of factors that may influence this decision. They include:

- Has one side won?
- How many ply have we already explored?
- How promising is this path?
- How much time is left?
- How stable is the configuration?

For the general MINIMAX procedure discussed here, we appeal to a function, DEEP-ENOUGH, which is assumed to evaluate all of these factors and to return TRUE if the search should be stopped at the current level and FALSE otherwise. Our simple implementation of DEEP-ENOUGH will take two parameters, *Position* and *Depth.* It will ignore its *Position* parameter and simply return TRUE if its *Depth* parameter exceeds a constant cutoff value.

One problem that arises in defining MINIMAX as a recursive procedure is that it needs to return not one but two results:

- The backed-up value of the path it chooses.
- The path itself. We return the entire path even though probably only the first element, representing the best move from the current position, is actually needed.

We assume that MINIMAX returns a structure containing both results and that we have two functions, VALUE and PATH, that extract the separate components.

Since we define the MINIMAX procedure as a recursive function, we must also specify how it is to be called initially. It takes three parameters, a board position, the current depth of the search, and the player to move. So the initial call to compute the best move from the position CURRENT should be

```
MINIMAX(CURRENT,0,PLAYER-ONE)
```

if PLAYER-ONE is to move, or

```
MINIMAX(CURRENT,0,PLAYER-TWO)
```

if PLAYER-TWO is to move.

[2] This may be a bit confusing, but it need not be. In all the examples in this chapter so far (including Fig. 12.2 and 12.3), we have assumed that all values of STATIC are from the point of view of the initial (maximizing) player. It turns out to be easier when defining the algorithm, though, to let STATIC alternate perspectives so that we do not need to write separate procedures for the two levels. It is easy to modify STATIC for this purpose; we merely compute the value *of Position* from PLAYER-ONE's perspective, then invert the value if STATIC's parameter is PLAYER-TWO.

Algorithm: MINIMAX(Position, Depth, Player)

```
1. If DEEP-ENOUGH(Position, Depth), then return the structure
      VALUE = STATIC(Position, Player);
      PATH = nil
   This indicates that there is no path from this node and that its value is that determined by the static evaluation function.
2. Otherwise, generate one more ply of the tree by calling the function MOVE-GEN(Position Player) and setting SUCCESSORS to the list it returns.
3. If SUCCESSORS is empty, then there are no moves to be made, so return the same structure that would have been returned if DEEP-ENOUGH had returned true.
4. If SUCCESSORS is not empty, then examine each element in turn and keep track of the best one. This is done as follows.
   Initialize BEST-SCORE to the minimum value that STATIC can return. It will be updated to reflect the hest score that can be achieved by an element of SUCCESSORS.
   For each element SUCC of SUCCESSORS, do the following:
   (a) Set RESULT-SUCC to
          MINIMAX(SUCC, Depth + 1, OPPOSITE(Player))
       This recursive call to MINIMAX will actually carry out the exploration of SUCC.
   (b) Set NEW-VALUE to - VALUE(RESULT-SUCC). This will cause it to reflect the merits of the position from the opposite perspective from that of the next lower level.
   (c) If NEW-VALUE > BEST-SCORE, then we have found a successor that is better than any that have been examined so far. Record this by doing the following:
        (i) Set BEST-SCORE to NEW-VALUE.
       (ii) The best known path is now from CURRENT to SUCC and then on to the appropriate path down from SUCC as determined hy the recursive call to MINIMAX. So set BEST-PATH to the result of attaching SUCC to the front of PATH(RESULT-SUCC).
5. Now that all the successors have been examined, we know the value of Position as well as which path to take from it. So return the structure
      VALUE = BEST-SCORE
      PATH = BEST-PATH
```

When the initial call to MINIMAX returns, the best move from CURRENT is the first element on PATH. To see how this procedure works, you should trace its execution for the game tree shown in Fig. 12.2.

The MINIMAX procedure just described is very simple. But its performance can be improved significantly with a few refinements. Some of these are described in the next few sections.

12.3 ADDING ALPHA-BETA CUTOFFS

Recall that the minimax procedure is a depth-first process. One path is explored as far as time allows, the static evaluation function is applied to the game positions at the last step of the path, and the value can then be passed up the path one level at a time. One of the good things about depth-first procedures is that their efficiency can often be improved by using branch-and-bound techniques in which partial solutions that are clearly worse than known solutions can be abandoned early. We described a straightforward application of this technique to the traveling salesman problem in Section 2.2.1. For that problem, all that was required was storage of the length of the best path found so far. If a later partial path outgrew that bound, it was abandoned.

But just as it was necessary to modify our search procedure slightly to handle both maximizing and minimizing players, it is also necessary to modify the branch-and-bound strategy to include two bounds, one for each of the players. This modified strategy is called *alpha-beta pruning*. It requires the maintenance of two threshold values, one representing a lower bound on the value that a maximizing node may ultimately be assigned (we call this *alpha*) and another representing an upper bound on the value that a minimizing node may be assigned (this we call *beta*).

To see how the alpha-beta procedure works, consider the example shown in Fig. 12.4.[3] After examining node F, we know that the opponent is guaranteed a score of –5 or less at C (since the opponent is the minimizing player). But we also know that we are guaranteed a score of 3 or greater at node A, which we can achieve if we move to B. Any other move that produces a score of less than 3 is worse than the move to B, and we can ignore it. After examining only F, we are sure that a move to C is worse (it will be less than or equal to –5) regardless of the score of node G. Thus we need not bother to explore node G at all. Of course, cutting out one node may not appear to justify the expense of keeping track of the limits and checking them, but if we were exploring this tree to six ply, then we would have eliminated not a single node but an entire tree three ply deep.

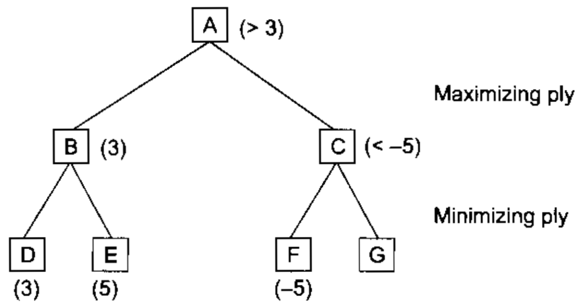


Fig. 12.4 *An Alpha Cutoff*

To see how the two thresholds, alpha and beta, can both be used, consider the example shown in Fig. 12.5 In searching this tree, the entire subtree headed by B is searched, and we discover that at A we can expect a score of at least 3. When this alpha value is passed down to F, it will enable us to skip the exploration of L. Let's see why. After K is examined, we see that I is guaranteed a maximum score of 0, which means that F is

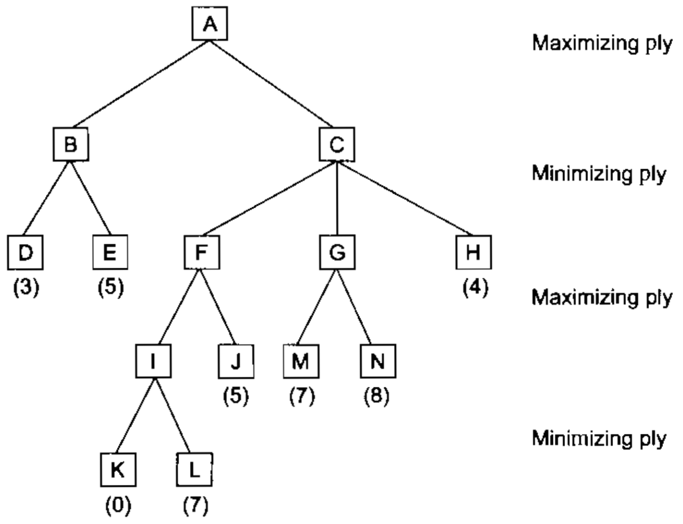


Fig. 12.5 *Alpha and Beta Cutoffs*

[3]In this figure, we return to the use of a single STATIC function from the point of view of the maximizing player.

guaranteed a minimum of 0. But this is less than alpha's value of 3, so no more branches of I need be considered. The maximizing player already knows not to choose to move to C and then to I since, if that move is made, the resulting score will be no better than 0 and a score of 3 can be achieved by moving to B instead. Now let's see how the value of beta can be used. After cutting off further exploration of I, J is examined, yielding a value of 5, which is assigned as the value of F (since it is the maximum of 5 and 0). This value becomes the value of beta at node C. It indicates that C is guaranteed to get a 5 or less. Now we must expand G. First M is examined and it has a value of 7, which is passed back to G as its tentative value. But now 7 is compared to beta (5). It is greater, and the player whose turn it is at node C is trying to minimize. So this player will not choose G, which would lead to a score of at least 7, since there is an alternative move to F, which will lead to a score of 5. Thus it is not necessary to explore any of the other branches of G.

From this example, we see that at maximizing levels, we can rule out a move early if it becomes clear that its value will be less than the current threshold, while at minimizing levels, search will be terminated if values that are greater than the current threshold are discovered. But ruling out a possible move by a maximizing player actually means cutting off the search at a minimizing level. Look again at the example in Fig. 12.4. Once we determine that C is a bad move from A, we cannot bother to explore G, or any other paths, at the minimizing level below C. So the way alpha and beta are actually used is that search at a minimizing level can be terminated when a value less than alpha is discovered, while a search at a maximizing level can be terminated when a value greater than beta has been found. Cutting off search at a maximizing level when a high value is found may seem counterintuitive at first, but if you keep in mind that we only get to a particular node at a maximizing level if the minimizing player at the level above chooses it, then it makes sense.

Having illustrated the operation of alpha-beta pruning with examples, we can now explore how the MINIMAX procedure described in Section 12.2 can be modified to exploit this technique. Notice that at maximizing levels, only beta is used to determine whether to cut off the search, and at minimizing levels only alpha is used. But at maximizing levels alpha must also be known since when a recursive call is made to MINIMAX, a minimizing level is created, which needs access to alpha. So at maximizing levels alpha must be known not so that it can be used but so that it can be passed down the tree. The same is true of minimizing levels with respect to beta. Each level must receive both values, one to use and one to pass down for the next level to use.

The MINIMAX procedure as it stands does not need to treat maximizing and minimizing levels differently since it simply negates evaluations each time it changes levels. It would be nice if a comparable technique for handling alpha and beta could be found so that it would still not be necessary to write separate procedures for the two players. This turns out to be easy to do. Instead of referring to alpha and beta, MINIMAX uses two values, USE-THRESH and PASS-THRESH. USE-THRESH is used to compute cutoffs. PASS-THRESH is merely passed to the next level as its USE-THRESH. Of course, USE-THRESH must also be passed to the next level, but it will be passed as PASS-THRESH so that it can be passed to the third level down as USE-THRESH again, and so forth. Just as values had to be negated each time they were passed across levels, so too must these thresholds be negated. This is necessary so that, regardless of the level of the search, a test for greater than will determine whether a threshold has been crossed. Now there need still be no difference between the code required at maximizing levels and that required at minimizing ones.

We have now described how alpha and beta values are passed down the tree. In addition, we must decide how they are to be set. To see how to do this, let's return first to the simple example of Fig. 12,4. At a maximizing level, such as that of node A, alpha is set to be the value of the best successor that has yet been found. (Notice that although at maximizing levels it is beta that is used to determine cutoffs, it is alpha whose new value can be computed. Thus at any level, USE-THRESH will be checked for cutoffs and PASS-THRESH will be updated to be used later.) But if the maximizing node is not at the top of the tree, we must also consider the alpha value that was passed down from a higher node. To see how this works, look again at Fig. 12.5 and consider what happens at node F. We assign the value 0 to node I on the basis, of examining node K. This is so far the best successor of F. But from an earlier exploration of the subtree headed by B, alpha was set to 3 and passed down from A to F. Alpha should not be reset to 0 on the basis of node I. It should stay as 3 to reflect the best move found so far in the entire tree. Thus we see that at a maximizing level, alpha should be set to either the value it had at the next-highest maximizing level or the best value found at this level, whichever is greater. The corresponding statement can be made about beta at minimizing levels. In fact, what we want to say is that at any level, PASS-THRESH should always be the maximum of the value it inherits from above and the best move found at its level. If PASS-THRESH is updated, the new value should be propagated both down to lower levels and back up to higher ones so that it always reflects the best move found anywhere in the tree.

At this point, we notice that we are doing the same thing in computing PASS-THRESH that we did in MINIMAX to compute BEST-SCORE. We might as well eliminate BEST-SCORE and let PASS-THRESH serve in its place.

With these observations, we are in a position to describe the operation of the function MINIMAX-A-B, which requires four arguments, *Position, Depth, Use-Thresh,* and *Pass-Thresh.* The initial call, to choose a move for PLAYER-ONE from the position CURRENT, should be

```
MINIMAX-A-B(CURRENT,
            0,
            PLAYER-ONE,
            maximum value STATIC can compute,
            minimum value STATIC can compute)
```

These initial values for *Use-Thresh* and *Pass-Thresh* represent the worst values that each side could achieve.

Algorithm: MINIM AX-A-B(Position, Depth, Player, Use-Thresh, Pass-Thresh)

```
1. If DEEP-ENOUGH(Position, Depth), then return the structure
      VALUE = STATIC(Position, Player);
      PATH = nil
2. Otherwise, generate one more ply of the tree by calling the function MOVE- GEN(Position, Player)
   and setting SUCCESSORS to the list it returns.
3. If SUCCESSORS is empty, there are no moves to be made; return the same structure that would have
   been returned if DEEP-ENOUGH had returned TRUE.
4. If SUCCESSORS is not empty, then go through it, examining each element and keeping track of the
   best one. This is done as follows.
   For each element SUCC of SUCCESSORS:
   (a) Set RESULT-SUCC to
       MINIMAX-A-B(SUCC, Depth + 1, OPPOSITE(Player),
       –Pass-Thresh, –Use-Thresh).
   (b) Set NEW-VALUE to –VALUE(RESULT-SUCC).
   (c) If NEW-VALUE > Pass-Thresh, then we have found a successor that is better than any that have
       been examined so far. Record this by doing the following.
        (i) Set Pass-Thresh to NEW-VALUE.
       (ii) The best known path is now from CURRENT to SUCC and then on to the appropriate path
            from SUCC as determined by the recursive call to MINIMAX-A-B. So set BEST-PATH to
            the result of attaching SUCC to the front of PATH(RESULT-SUCC).
```

(d) If *Pass-Thresh* (reflecting the current best value) is not better than *Use-Thresh*, then we should stop examining this branch. But both thresholds and values have been inverted. So if *Pass-Thresh* >= *Use-Thresh*, then return immediately with the value

 VALUE = *Pass-Thresh*
 PATH = BEST-PATH

5. Return the structure
 VALUE = *Pass-Thresh*
 PATH = BEST-PATH

The effectiveness of the alpha-beta procedure depends greatly on the order in which paths are examined. If the worst paths are examined first, then no cutoffs at all will occur. But, of course, if the best path were known in advance so that it could be guaranteed to be examined first, we would not need to bother with the search process. If, however, we knew how effective the pruning technique is in the perfect case, we would have an upper bound on its performance in other situations. It is possible to prove that if the nodes are perfectly ordered, then the number of terminal nodes considered by a search to depth *d* using alpha-beta pruning is approximately equal to twice the number of terminal nodes generated by a search to depth *d*/2 without alpha-beta [Knuth and Moore, 1975]. A doubling of the depth to which the search can be pursued is a significant gain. Even though all of this improvement cannot typically be realized, the alpha-beta technique is a significant improvement to the minimax search procedure.- For a more detailed study of the average branching factor of the alpha-beta procedure, see Baudet [1978] and Pearl [1982].

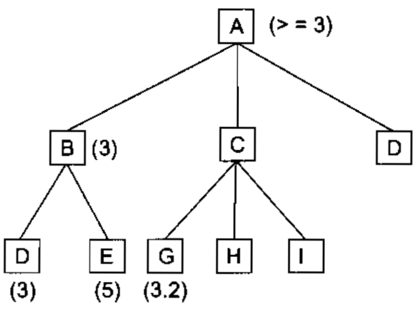


Fig. 12.6 *A Futility Cutoff*

The idea behind the alpha-beta procedure can be extended to cut off additional paths that appear to be at best only slight improvements over paths that have already been explored. In step 4(*d*), we cut off the search if the path we were exploring was not better than other paths already found. But consider the situation shown in Fig. 12.6. After examining node G, we see that the best we can hope for if we make move C is a score of 3.2. We know that if we make move B we are guaranteed a score of 3. Since 3.2 is only very slightly better than 3, we should perhaps terminate our exploration of C now. We could then devote more time to exploring other parts of the tree where there may be more to gain. Terminating the exploration of a subtree that offers little possibility for improvement over other known paths is called a *futility cutoff.*

12.4 ADDITIONAL REFINEMENTS

In addition to alpha-beta pruning, there are a variety of other modifications to the minimax procedure that can also improve its performance. Four of them are discussed briefly in this section, and we discuss one other important modification in the next section.

12.4.1 Waiting for Quiescence

As we suggested above, one of the factors that should sometimes be considered in determining when to stop going deeper in the search tree is whether the situation is relatively stable. Consider the tree shown in Fig. 12.7. Suppose that when node B is expanded one more level, the result is that shown in Fig. 12.8. When we looked one move ahead, our estimate of the worth of B changed drastically. This might happen, for example, in the middle of a piece exchange. The opponent has significantly improved the immediate appearance of his or her position by initiating a piece exchange. If we stop exploring the tree at this level, we assign the value −4 to B and therefore decide that B is not a good move.

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 flat 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

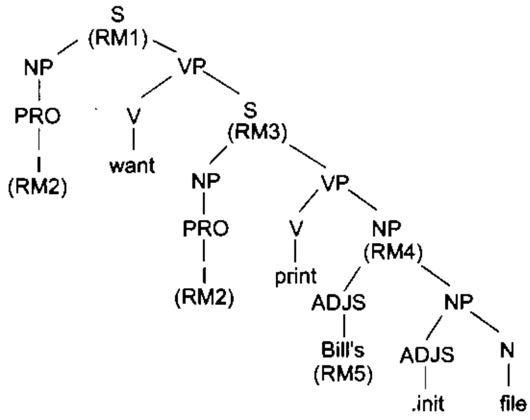


Fig. 15.2 *The Result of Syntactic Analysis of "I want to print Bill's .init file."*

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 /wsmith/stuff.init
```

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.

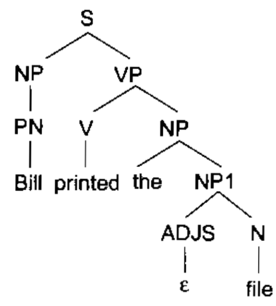


Fig. 15.7 *A Parse Tree for a Sentence*

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.

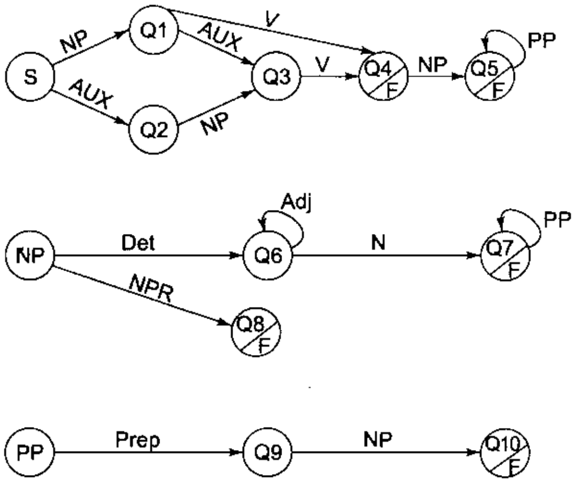


Fig. 15.8 *An ATN Network for a Fragment of English*

```
(S/     (PUSH NP/T
            (SETR SUBJ *)
            (SETR TYPE (QUOTE DCL))
            (TO Q1))
        (CAT AUX T
            (SETR AUX *)
            (SETR TYPE (OUOTE O))
            (TO Q2)))
(QI     (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 Ql, 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 Ql, 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:

$NP \rightarrow DET\ N$

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 *G*1 or *G*2 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 *G*1 or *G*2, 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 tbe 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.

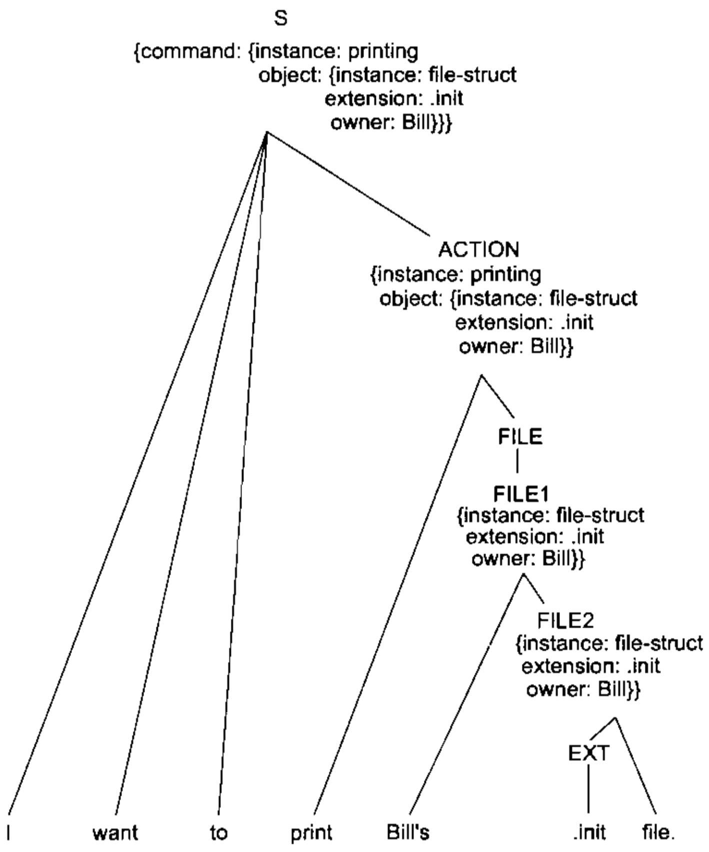


Fig. 15.11 *The Result of Parsing with a Semantic Grammar*

- 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.

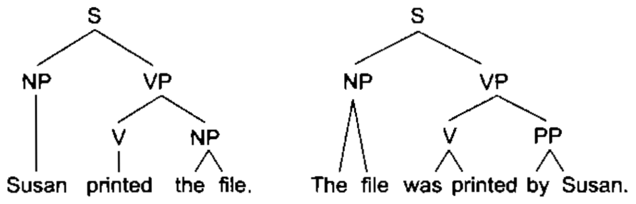


Fig. 15.12 *Syntactic Parses of an Active and a Passive Sentence*

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.

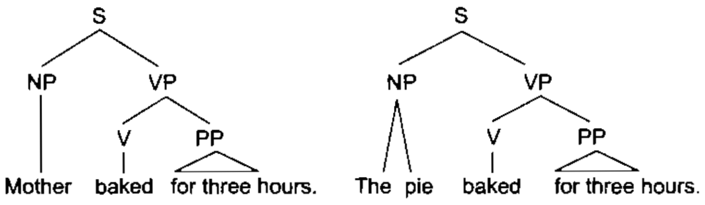


Fig. 15.13 *Syntactic Parses of Two Similar Sentences*

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.

These rules can be applied in reverse by a parser to determine the underlying case structure from the 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:

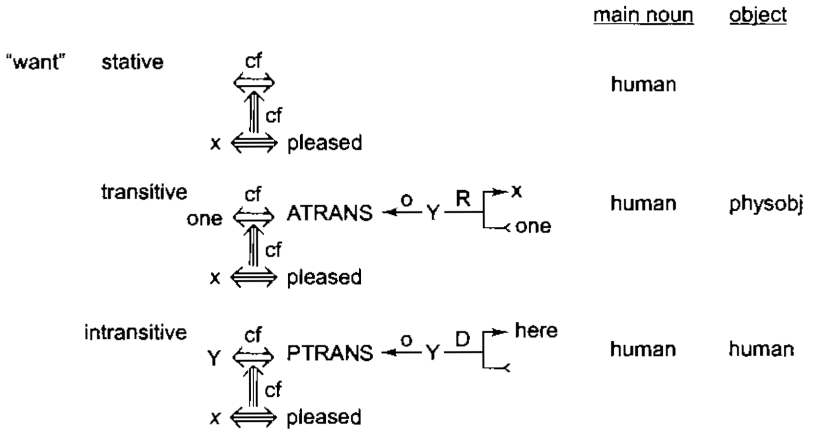


Fig. 15.15 *The Verb-ACT Dictionary*

- 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.

the structure shown in Fig. 15.16 would be built.

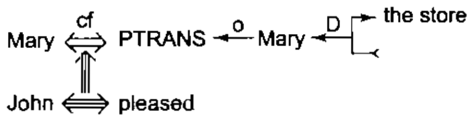


Fig. 15.16 *A CD Structure*

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.

Since peacocks are animate, they would be acceptable as additional actors of the main verb, "went." Thus, 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.

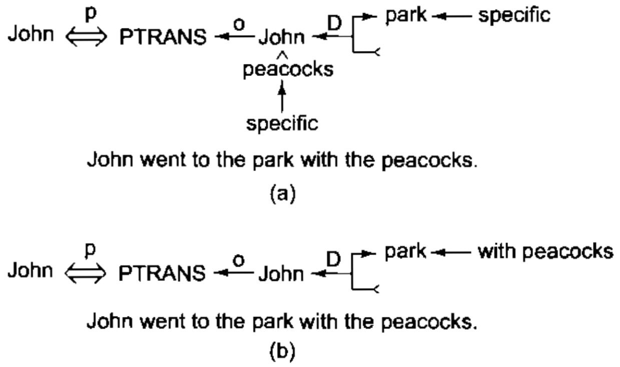


Fig. 15.17 *Two CD Interpretations of a Sentence*

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.

The third and fourth rules are examples of modifier rules. Like the verb rules, they too must specify both 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, $\{x \mid \text{John eats } x \text{ on Friday}\}$)

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 person can be used to refer to things the person creates. Authoring is a kind of creating.
- 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.

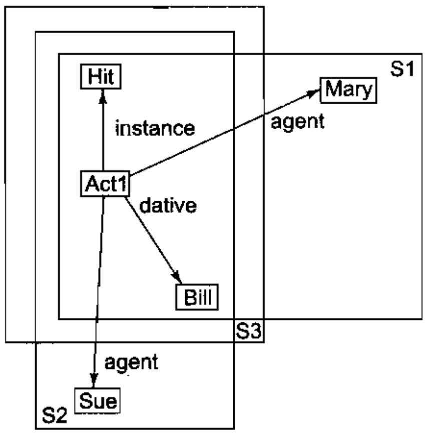


Fig. 15.20 *A Partitioned Semantic Net Showing Three Belief Spaces*

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?"

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?*

Given the word *however*, and the words *the* and *it* that could follow we can use the respective probabilities (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

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

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 $P(w_n \mid (w_1.w_2 \ldots w_{n-1}))$ can be approximated to the product of all $P(w_i \mid w_{i-1})$ 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 may define a trigram wherein given a sequence of two words we 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:

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

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) Non-Word Error Detection:

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:

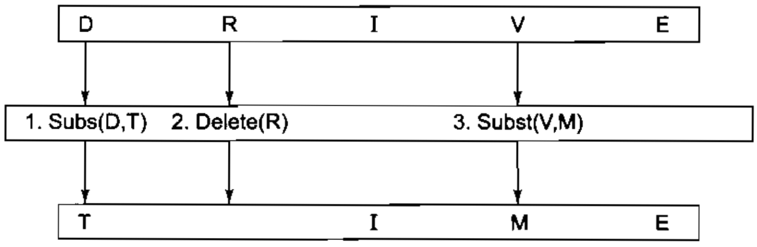
This process focuses on the correction of an isolated non-word by finding its nearest and meaningful word and makes an attempt to rectify the error. It thus transforms the word *soper* into *super* by some means but without looking into the context.

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 thc 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 character of word, Code_1, Code_2, Code_3. Table 15.5 shows the Soundex codes 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."