MODULE-III

Knowledge Representation

It is a medium for human expression. An intelligent system must have Knowledge Representations that can be interpreted by humans. We need to be able to encode information in the knowledge base without significant effort. We need to be able to understand what the system knows and how it draws its conclusions.

It is necessary to represent the computer's knowledge of the world by some kind of data structures in the machine's memory. Traditional computer programs deal with large amounts of data that are structured in simple and uniform ways. A.I. programs need to deal with complex relationships, reflecting the complexity of the real world.

Several kinds of knowledge need to be represented:

- **Factual Data:** Known facts about the world.
- **General Principles:** ``Every dog is a mammal."
- **Hypothetical Data:** The computer must consider hypotheticals in order to reason about the effects of actions that are being contemplated.

Types

1) Logic (prepositional, predicate)
2) Network representation
    - Semantic nets
3) Structured representation
    - Frames

Semantic Nets

Semantic Nets were invented by Quillian in 1969. Quillian was a Psychologist (at UMich) and was trying to define the structure of human knowledge. Semantic Nets are about relations between concepts. Semantic Nets use a graph structure so that concepts are nodes in the graph.

An important feature of human memory is the high number of connections or associations between the different pieces of information contained in it. A semantic network is one of the knowledge representation languages based on this intuition. The basic idea of a semantic network representation is very simple: There are two types of primitive, nodes and links or arcs. Links are unidirectional connections between nodes. Nodes correspond to objects, or classes of objects, in the world, whereas links correspond to relationships between these objects.

Semantic nets are usually used to represent static, taxonomic, concept dictionaries. Semantic nets were originally designed as a way to represent the meanings of English words. 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. Major problem with semantic nets is that although the name of this knowledge representation language is semantic nets, there is not, ironically, clear semantics of the various network representations. A fragment of a typical semantic net is shown in Figure 1 below.

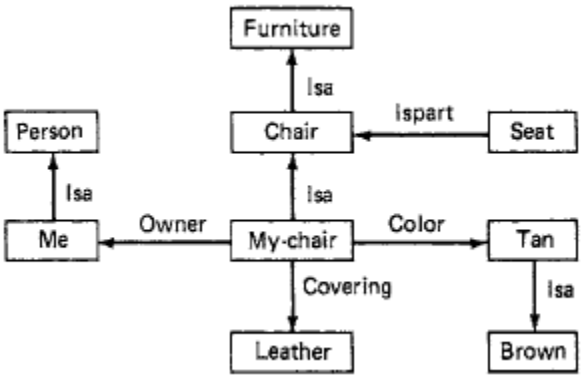


Figure 1: A Semantic Network

It is useful to think about semantic nets using a graphical notation, as shown in the figure. Of course, they cannot be represented inside a program that way. Instead, they are usually represented using some kind of attribute-value memory structure.

Notice that each property stores a one-way link, such as the arc from MY-CHAIR to ME. To store bidirectional links, it is necessary to store each half separately. So if we wanted to be able to answer the question "What do I own?" without searching the entire net, we would need arcs from ME to all the nodes that connect to ME via OWNER arcs. Since it means something different to go from MY-CHAIR to ME than it does to go from ME to MY-CHAIR, we need a new kind of arc, which we can call OWNED. It is, of course, more efficient to store only one-way arcs. But then it is difficult to form inferences that go in the missing direction.

For each system, it is necessary to decide what kind of inferences will be needed and then to design the links appropriately; this is the same problem that designers of any database must solve when they decide what fields to index on.

Important semantic relations:

- Meronymy (A is part of B, i.e. B has A as a part of itself)
- Holonymy (B is part of A, i.e. A has B as a part of itself)
- Hyponymy (or troponymy) (A is subordinate of B; A is kind of B)
- Hypernymy (A is superordinate of B)
- Synonymy (A denotes the same as B)
- Antonymy (A denotes the opposite of B)

Representing Non-binary Predicates

Semantic nets can be used to represent relationships that would appear as two-place predicates in predicate logic. For example, some of the arcs from Figure 1 could be represented in logic as

ISA(chair, furniture)

ISA(me,person)

COVERING(my-chair,leather)

COLOR(my-chair ,tan)

We have already seen that many one-place predicates in logic can be thought of as two-place predicates using some very general-purpose predicates, such as ISA. But the knowledge expressed by other predicates can also be expressed in semantic nets. So, for example,

MAN(Marcus) could be rewritten as

ISA(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 thatSCORE(red blue (17 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 Figure 2.

Figure 2: A Semantic Net for an N-Place Predicate

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

John gave the book to Mary could be represented by the network shown in Figure 3.

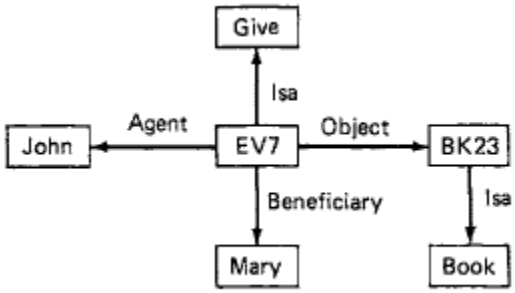


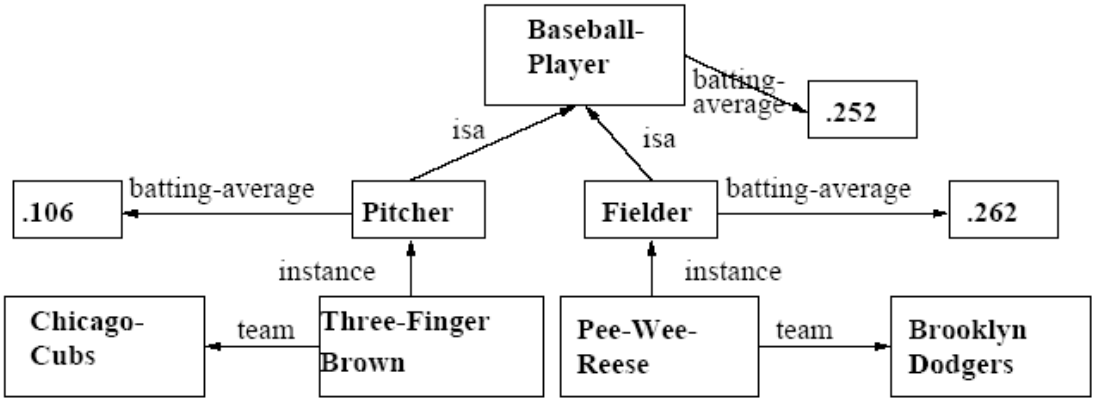
Figure 3: A Semantic Net Representing a Sentence

The node labeled BK23 represents the particular book that was referred to by the phrase the book. Note that we are again using a case grammar analysis of the sentence. Semantic nets have been used to represent a variety of kinds of knowledge in a variety of different programs.

Reasoning with the Knowledge

As with any other mechanism for representing knowledge, the power of semantic nets lies in the ability of programs to manipulate them to solve problems. 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 was called intersection search. Using this process, it is possible to use the network of figure below to answer questions such as

Question: “What is the relation between Chicago cubs and Brooklyn Dodgers?”

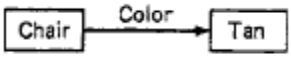


Answer: “They are both teams of baseball players.”

More recent applications of semantic nets have used more directed search procedures to answer specific questions. Although in principle there are no restrictions on the way information can be represented in the networks, these search procedures can only be effective if there is consistency in what each node and each link mean. Let's look at a couple of examples of this.

There is a difference between a concept (such as GAME or GIVE) and instances of that concept (such as G23 or EV7). Whereas my chair can get wet, the concept *chair* cannot. If information about specific instances is stored at the node representing the concept, it is not possible to differentiate among multiple instances of the same concept. For example, we could represent the fact that

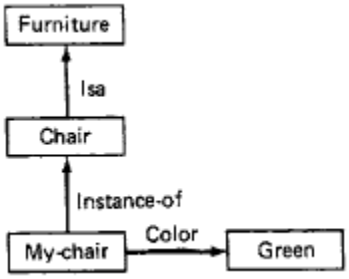
My chair is tan.using the semantic net as



But we would then have a hard time representing the additional fact that Mary's chair is green. To avoid this problem, links from the concept node should usually be used to describe properties of all (or most) instances of the concept, while links from an instance node describe properties of the individual instance. It is important to make explicit this difference between a node representing a class of objects and a node representing an instance of a class.

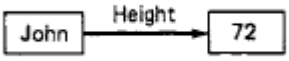
We have been using ISA links to represent hierarchical relationships between concept nodes, and sometimes also to relate nodes representing specific objects to their associated concepts. But in order to maintain the distinction between concept nodes and instance nodes, we need to introduce a new kind of link by which instance nodes can be

connected to the concepts that describe them. We will call this new link INSTANCE-OF. An example of it occurs in the following fragment of a network:

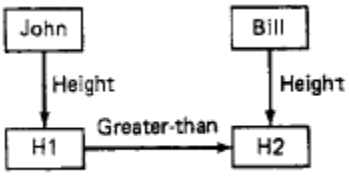


Sometimes it is also important to distinguish between a node representing the canonical instance of a concept and a node representing the set of all instances of it. For example, the set of all Americans is of size two and a quarter million, while the typical American is of size five and a half feet. There is a difference between a link that defines a new entity and one that relates two existing entities. For example

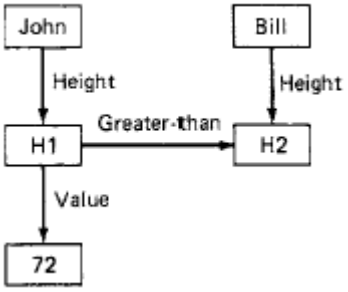
John is 6 ft tall can be represented using semantic net as below



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 HI and H2 represent the new concepts 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 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 represent the fact that John is 6 feet tall and that he is taller than Bill by the net



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.

Partitioned Semantic Nets

One of the major problems with the use of semantic nets to represent knowledge is how to handle quantification. One way of solving the problem is to partition the semantic net into a hierarchical set of spaces, each of which corresponds to the scope of one or more variables. To see how this works, consider first the simple net shown in Figure 4(a). This net corresponds to the statement

The dog bit the postman.

The nodes DOGS, BITE, and POSTMEN represent the classes of dogs, biting, and postmen, respectively, while the nodes D, B, and P represent a particular dog, a particular biting, and a particular postman. This fact can easily be represented by a single net with no partitioning.

But now suppose that we want to represent the fact that Every dog has bitten a postman.

or, in logic $\forall$ x dog(x)$\rightarrow$($\exists$y postman(y) $\wedge$ bite(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 Figure 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 V 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 p, are understood to be existentially quantified. In other words, for every dog d, there exists a biting event b, and a postman p, such that d is the assailant of b and p is the victim.

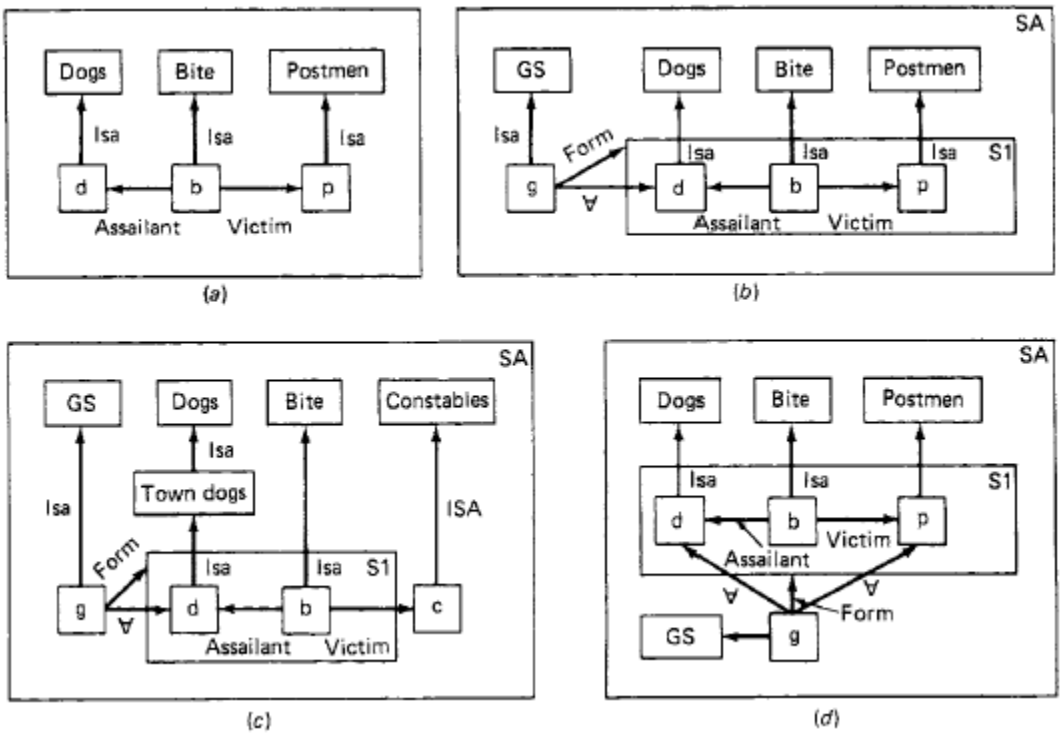


Figure 4: Using Partitioned Semantic Nets

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

Every dog in town has bitten the constable.

The representation of this sentence is shown in Figure 7-7(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, non partitioned net.

Figure 4(d) shows how yet another similar sentence:Every dog has bitten every postman.

In this case, g has two V links, one pointing to d, which represents any dog, and one pointing to p, representing any postman. The spaces of a partitioned semantic net are related to each other by an inclusion hierarchy. For example, in Figure 4(d), space S1 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 Figure 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 traversing ISA links, we would not find d, since it and the link to it are in the space S1, 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. Partitioned semantic nets have a variety of other uses besides their ability to encode quantification. Semantic nets have been widely used to represent a variety of kinds of knowledge.

Advantages of Semantic nets

- Easy to visualize
- Formal definitions of semantic networks have been developed.
- Related knowledge is easily clustered.
- Efficient in space requirements
  - Objects represented only once
  - Relationships handled by pointers

Semantic nets have two significant advantages over unstructured sets of logic clauses:

• Economy: each property only needs to be represented once: friendly to people for cat, and so on down to long tail for Maine Coon and orange for Charles Douglas. Imagine what your system would look like if, for every single individual Maine Coon cat, you had to write out all the information about Maine Coons, pedigree cats, cats, ... and so on up to animals (and beyond).

• Significant generalization: in a large system like that, it might not be easy to see that all the Maine Coons had long tails, and even if you did see it, you might think it was a coincidence. By making long tail a property of the class Maine Coon, you make clear both the fact, and that it is significant.

Disadvantages of Semantic nets

- Inheritance (particularly from multiple sources and when exceptions in inheritance are wanted) can cause problems.
- Facts placed inappropriately cause problems.
- No standards about node and arc values

Uses of Semantic Nets

- Coding static world knowledge
- Built-in fast inference method (inheritance)

4.3 ISSUES IN KNOWLEDGE REPRESENTATION

Before embarking on a discussion of specific mechanisms that have been used to represent various kinds of real-world knowledge, we need briefly to discuss several issues that cut across all of them:

- Are any attributes of objects so basic that they occur in almost every problem domain? If there are, we need to make sure that they are handled appropriately in each of the mechanisms we propose. If such attributes exist, what are they?
- Are there any important relationships that exist among attributes of objects?

- At what level should knowledge be represented? Is there a good set of *primitives* into which all knowledge can be broken down? Is it helpful to use such primitives?
- How should sets of objects be represented?
- Given a large amount of knowledge stored in a database, how can relevant parts be accessed when they are needed?

We will talk about each of these questions briefly in the next five sections.

4.3.1 Important Attributes

There are two attributes that are of very general significance, and we have already seen their use: *instance* and *isa*. These attributes are important because they support property inheritance. They are called a variety of things in AI systems, but the names do not matter. What does matter is that they represent class membership and class inclusion and that class inclusion is transitive. In slot-and-filler systems, such as those described in Chapters 9 and 10, these attributes are usually represented explicitly in a way much like that shown in Fig. 4.5 and 4.6. In logic-based systems, these relationships may be represented this way or they may be represented implicitly by a set of predicates describing particular classes. See Section 5.2 for some examples of this.

4.3.2 Relationships among Attributes

The attributes that we use to describe objects are themselves entities that we represent. What properties do they have independent of the specific knowledge they encode? There are four such properties that deserve mention here:

- Inverses
- Existence in an *isa* hierarchy
- Techniques for reasoning about values
- Single-valued attributes

Inverses

Entities in the world are related to each other in many different ways. But as soon as we decide to describe those relationships as attributes, we commit to a perspective in which we focus on one object and look for binary relationships between it and others. Attributes are those relationships. So, for example, in Fig. 4.5, we used the attributes *instance*, *isa* and *team*. Each of these was shown in the figure with a directed arrow, originating at the object that was being described and terminating at the object representing the value of the specified attribute. But we could equally well have focused on the object representing the value. If we do that, then there is still a relationship between the two entities, although it is a different one since the original relationship was not symmetric (although some relationships, such as *sibling*, are). In many cases, it is important to represent this other view of relationships. There are two good ways to do this.

The first is to represent both relationships in a single representation that ignores focus. Logical representations are usually interpreted as doing this. For example, the assertion:

team(Pee-Wee-Reese, Brooklyn-Dodgers)

can equally easily be interpreted as a statement about Pee Wee Reese or about the Brooklyn Dodgers. How it is actually used depends on the other assertions that a system contains.

The second approach is to use attributes that focus on a single entity but to use them in pairs, one the inverse of the other. In this approach, we would represent the team information with two attributes:

- one associated with Pee Wee Reese:
  team = Brooklyn-Dodgers

- one associated with Brooklyn Dodgers:
  team-members = Pee-Wee-Reese,...

This is the approach that is taken in semantic net and frame-based systems. When it is used, it is usually accompanied by a knowledge acquisition tool that guarantees the consistency of inverse slots by forcing them to be declared and then checking each time a value is added to one attribute that the corresponding value is added to the inverse.

An Isa Hierarchy of Attributes

Just as there are classes of objects and specialized subsets of those classes, there are attributes and specializations of attributes. Consider, for example, the attribute *height*. It is actually a specialization of the more general attribute *physical-size* which is, in turn, a specialization of *physical-attribute*. These generalization-specialization relationships are important for attributes for the same reason that they are important for other concepts—they support inheritance. In the case of attributes, they support inheriting information about such things as constraints on the values that the attribute can have and mechanisms for computing those values.

Techniques for Reasoning about Values

Sometimes values of attributes are specified explicitly when a knowledge base is created. We saw several examples of that in the baseball example of Fig. 4.5. But often the reasoning system must reason about values it has not been given explicitly. Several kinds of information can play a role in this reasoning, including:

- Information about the type of the value. For example, the value of *height* must be a number measured in a unit of length.
- Constraints on the value, often stated in terms of related entities. For example, the age of a person cannot be greater than the age of either of that person's parents.
- Rules for computing the value when it is needed. We showed an example of such a rule in Fig. 4.5 for the *bats* attribute. These rules are called *backward* rules. Such rules have also been called *if-needed rules*.
- Rules that describe actions that should be taken if a value ever becomes known. These rules are called *forward* rules, or sometimes *if-added rules*.

We discuss forward and backward rules again in Chapter 6, in the context of rulebased knowledge representation.

Single-Valued Attributes

A specific but very useful kind of attribute is one that is guaranteed to take a unique value. For example, a baseball player can, at any one time, have only a single height and be a member of only one team. If there is already a value present for one of these attributes and a different value is asserted, then one of two things has happened. Either a change has occurred in the world or there is now a contradiction in the knowledge base that needs to be resolved. Knowledge-representation systems have taken several different approaches to providing support for single-valued attributes, including:

- Introduce an explicit notation for temporal interval. If two different values are ever asserted for the same temporal interval, signal a contradiction automatically.
- Assume that the only temporal interval that is of interest is now. So if a new value is asserted, replace the old value.
- Provide no explicit support. Logic-based systems are in this category. But in these systems, knowledge-base builders can add axioms that state that if an attribute has one value then it is known not to have all other values.

4.3.3 Choosing the Granularity of Representation

Regardless of the particular representation formalism we choose, it is necessary to answer the question "At what level of detail should the world be represented?" Another way this question is often phrased is "What should be our primitives?" Should there be a small number of low-level ones or should there be a larger number covering a range of granularities? A brief example illustrates the problem. Suppose we are interested in the following fact:

John spotted Sue.

We could represent this as[1]

spotted(agent(John),
object(Sue))

Such a representation would make it easy to answer questions such as:

Who spotted Sue?

But now suppose we want to know:

Did John see Sue?

The obvious answer is "yes," but given only the one fact we have, we cannot discover that answer. We could, of course, add other facts, such as

$$spotted(x, y) \rightarrow saw(x, y)$$

We could then infer the answer to the question.

An alternative solution to this problem is to represent the fact that spotting is really a special type of seeing explicitly in the representation of the fact. We might write something such as

saw(agent(John),
object(Sue),
timespan(briefly))

In this representation, we have broken the idea of *spotting* apart into more primitive concepts *of seeing* and *timespan*. Using this representation, the fact that John saw Sue is immediately accessible. But the fact that he spotted her is more difficult to get to.

The major advantage of converting all statements into a representation in terms of a small set of primitives is that the rules that are used to derive inferences from that knowledge need be written only in terms of the primitives rather than in terms of the many ways in which the knowledge may originally have appeared. Thus what is really being argued for is simply some sort of canonical form. Several AI programs, including those described by Schank and Abelson [1977] and Wilks [1972], are based on knowledge bases described in terms of a small number of low-level primitives.

Using Predicate logic

In this chapter, we begin exploring one particular way of representing facts — the language of logic. Other representational formalisms are discussed in later chapters. The logical formalism is appealing because it immediately suggests a powerful way of deriving new knowledge from old — mathematical deduction. In this formalism, we can conclude that a new statement is true by proving that it follows from the statements that are already known. Thus the idea of a proof, as developed in mathematics as a rigorous way of demonstrating the truth of an already believed proposition, can be extended to include deduction as a way of deriving answers to questions and solutions to problems.

One of the early domains in which AI techniques were explored was mechanical theorem proving, by which was meant proving statements in various areas of mathematics. For example, the Logic Theorist [Newell *et al.*, 1963] proved theorems from the first chapter of Whitehead and Russell's *Principia Mathematica* [1950]. Another theorem prover [Gelernter *et al.*, 1963] proved theorems in geometry. Mathematical theorem proving is still an active area of AI research. (See, for example, Wos *et al.* [1984].) But, as we show in this chapter, the usefulness of some mathematical techniques extends well beyond the traditional scope of mathematics. It turns out that mathematics is no different from any other complex intellectual endeavor in requiring both reliable deductive mechanisms and a mass of heuristic knowledge to control what would otherwise be a completely intractable search problem.

2. Marcus was a Pompeian.

   $Pompeian(Marcus)$

3. All Pompeians were Romans.

   $\forall x : Pompeian(x) \rightarrow Roman(x)$

4. Caesar was a ruler.

   $ruler(Caesar)$

   Here we ignore the fact that proper names are often not references to unique individuals, since many people share the same name. Sometimes deciding which of several people of the same name is being referred to in a particular statement may require a fair amount of knowledge and reasoning.

5. All Romans were either loyal to Caesar or hated him.

   $\forall x : Roman(x) \rightarrow loyalto(x, Caesar) \vee hate(x, Caesar)$

   In English, the word "or" sometimes means the logical *inclusive-or* and sometimes means the logical *exclusive-or* (XOR). Here we have used the inclusive interpretation. Some people will argue, however, that this English sentence is really stating an,exclusive-or. To express that, we would have to write:

   $\forall x : Romman(x) \rightarrow [(loyal\ to(x, Caesar) \vee hate(x, Caesar)) \wedge$
   $\neg(loyalto(x, Caesar) \wedge hate(x,Caesar))]$

6. Everyone is loyal to someone.

   $\forall x : \rightarrow y : loyalto(x,y)$

   A major problem that arises when trying to convert English sentences into logical statements is the scope of quantifiers. Does this sentence say, as we have assumed in writing the logical formula above, that for each person there exists someone to whom he or she is loyal, possibly a different someone for everyone? Or does it say that there exists someone to whom everyone is loyal (which would be written as $\exists y : \forall x : loyalto(x,y)$)? Often only one of the two interpretations seems likely, so people tend to favor it.

7. People only try to assassinate rulers they are not loyal to.

   $\forall x : \forall y : person(x) \wedge ruler(y) \wedge tryassassinate(x,y) \rightarrow \neg\ loyalto(x,y)$

   This sentence, too, is ambiguous. Does it mean that the only rulers that people try to assassinate are those to whom they are not loyal (the interpretation used here), or does it mean that the only thing people try to do is to assassinate rulers to whom they are not loyal?

   In representing this sentence the way we did, we have chosen to write "try to assassinate" as a single predicate. This gives a fairly simple representation with which we can reason about trying to assassinate. But using this representation, the connections between trying to assassinate and trying to do other things and between trying to assassinate and actually assassinating could not be made easily. If such connections were necessary, we would need to choose a different representation.

8. Marcus tried to assassinate Caesar.

   $tryassassinate\ (Marcus, Caesar)$

5.1 REPRESENTING SIMPLE FACTS IN LOGIC

Let's first explore the use of propositional logic as a way of representing the sort of world knowledge that an AI system might need. Propositional logic is appealing because it is simple to deal with and a decision procedure for it exists. We can easily represent real-world facts as logical *propositions* written as *well-formed formulas* (*wff's*) in propositional logic, as shown in Fig. 5.1. Using these propositions, we could, for example, conclude from the fact that it is raining the fact that it is not sunny. But very quickly we run up against the limitations of propositional logic. Suppose we want to represent the obvious fact stated by the classical sentence

It is raining.
RAINING

It is sunny.
SUNNY

It is windy.
WINDY

If it is raining, then it is not sunny.
RAINING $\rightarrow \neg$ *SUNNY*

Fig. 5.1 *Some Simple Facts in Prepositional Logic*

Socrates is a man.

We could write:

SOCRATESMAN

But if we also wanted to represent

Plato is a man.

we would have to write something such as:

PLATOMAN

which would be a totally separate assertion, and we would not be able to draw any conclusions about similarities between Socrates and Plato. It would be much better to represent these facts as:

MAN(SOCRATES)
MAN(PLATO)

since now the structure of the representation reflects the structure of the knowledge itself. But to do that, we need to be able to use predicates applied to arguments. We are in even more difficulty if we try to represent the equally classic sentence

All men are mortal.

We could represent this as:

MORTALMAN

But that fails to capture the relationship between any individual being a man and that individual being a mortal. To do that, we really need variables and quantification unless we are willing to write separate statements about the mortality of every known man.

So we appear to be forced to move to first-order predicate logic (or just predicate logic, since we do not discuss higher order theories in this chapter) as a way of representing knowledge because it permits representations of things that cannot reasonably be represented in propositional logic. In predicate logic, we can represent real-world facts as *statements* written as wff's.

But a major motivation for choosing to use logic at all is that if we use logical statements as a way of representing knowledge, then we have available a good way of reasoning with that knowledge. Determining the validity of a proposition in propositional logic is straightforward, although it may be computationally hard. So before we adopt predicate logic as a good medium for representing knowledge, we need to ask whether it also provides a good way of reasoning with the knowledge. At first glance, the answer is yes. It provides a way of deducing new statements from old ones. Unfortunately, however, unlike propositional logic, it does not possess a decision procedure, even an exponential one. There do exist procedures that will find a proof of a proposed theorem if indeed it is a theorem. But these procedures are not guaranteed to halt if the proposed statement is not a theorem. In other words, although first-order predicate logic is not decidable, it is semidecidable. A simple such procedure is to use the rules of inference to generate theorem's from the axioms in some orderly fashion, testing each to see if it is the one for which a proof is sought. This method is not particularly efficient, however, and we will want to try to find a better one.

Although negative results, such as the fact that there can exist no decision procedure for predicate logic, generally have little direct effect on a science such as AI, which seeks positive methods for doing things, this particular negative result is helpful since it tells us that in our search for an efficient proof procedure, we should be content if we find one that will prove theorems, even if it is not guaranteed to halt if given a nontheorem. And the fact that there cannot exist a decision procedure that halts on all possible inputs does not mean that there cannot exist one that will halt on almost all the inputs it would see in the process of trying to solve real problems. So despite the theoretical undecidability of predicate logic, it can still serve as a useful way of representing and manipulating some of the kinds of knowledge that an AI system might need.

Let's now explore the use of predicate logic as a way of representing knowledge by looking at a specific example. Consider the following set of sentences:

1. Marcus was a man.
2. Marcus was a Pompeian.
3. All Pompeians were Romans.
4. Caesar was a ruler.
5. All Romans were either loyal to Caesar or hated him.
6. Everyone is loyal to someone.
7. People only try to assassinate rulers they are not loyal to.
8. Marcus tried to assassinate Caesar.

The facts described by these sentences can be represented as a set of wff's in predicate logic as follows:

1. Marcus was a man.

   man(Marcus)

   This representation captures the critical fact of Marcus being a man. It fails to capture some of the information in the English sentence, namely the notion of past tense. Whether this omission is acceptable or not depends on the use to which we intend to put the knowledge. For this simple example, it will be all right.

From this brief attempt to convert English sentences into logical statements, it should be clear how difficult the task is. For a good description of many issues involved in this process, see Reichenbach [1947].

Now suppose that we want to use these statements to answer the question

Was Marcus loyal to Caesar?

It seems that using 7 and 8, we should be able to prove that Marcus was not loyal to Caesar (again ignoring the distinction between past and present tense). Now let's try to produce a formal proof, reasoning backward from the desired goal:

$\neg loyalto(Marcus, Caesar)$

In order to prove the goal, we need to use the rules of inference to transform it into another goal (or possibly a set of goals) that can in turn be transformed, and so on, until there are no unsatisfied goals remaining. This process may require the search of an AND-OR graph (as described in Section 3.4) when there are alternative ways of satisfying individual goals. Here, for simplicity, we show only a single path. Figure 5.2 shows an attempt to produce a proof of the goal by reducing the set of necessary but as yet unattained goals to the empty set. The attempt fails, however, since there is no way to satisfy the goal *person (Murcus)* with the statements we have available.

The problem is that, although we know that Marcus was a man, we do not have any way to conclude from that that Marcus was a person. We need to add the representation of another fact to our system, namely:

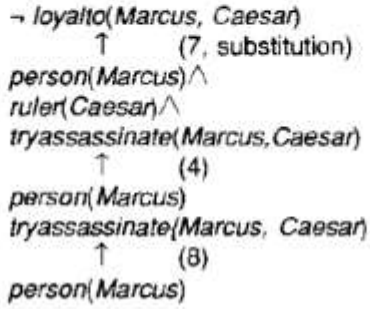


Fig. 5.2 *An Attempt to Prove ¬loyalto(Marcus,Caesar)*

9. All men are people.
   $\forall : man(x) \rightarrow person(x)$

Now we can satisfy the last goal and produce a proof that Marcus was not loyal to Caesar.

From this simple example, we see that three important issues must be addressed in the process of converting English sentences into logical statements and then using those statements to deduce new ones:

- Many English sentences are ambiguous (for example, 5, 6, and 7 above). Choosing the correct interpretation may be difficult.
- There is often a choice of how to represent the knowledge (as discussed in connection with 1, and 7 above). Simple representations are desirable, but they may preclude certain kinds of reasoning. The expedient representation for a particular set of sentences depends on the use to which the knowledge contained in the sentences will be put.

- Even in very simple situations, a set of sentences is unlikely to contain all the information necessary to reason about the topic at hand. In order to be able to use a set of statements effectively, it is usually necessary to have access to another set of statements that represent facts that people consider too obvious to mention. We discuss this issue further in Section 10.3.

An additional problem arises in situations where we do not know in advance which statements to deduce. In the example just presented, the object was to answer the question "Was Marcus loyal to Caesar?" How would a program decide whether it should try to prove

loyalto(Marcus, Caesar)
¬loyalto(Marcus, Caesar)

There are several things it could do. It could abandon the strategy we have outlined of reasoning backward from a proposed truth to the axioms and instead try to reason forward and see which answer it gets to. The problem with this approach is that, in general, the branching factor going forward from the axioms is so great that it would probably not get to either answer in any reasonable amount of time. A second thing it could do is use some sort of heuristic rules for deciding which answer is more likely and then try to prove that one first. If it fails to find a proof after some reasonable amount of effort, it can try the other answer. This notion of limited effort is important, since any proof procedure we use may not halt if given a nontheorem. Another thing it could do is simply try to prove both answers simultaneously and stop when one effort is successful. Even here, however, if there is not enough information available to answer the question with certainty, the program may never halt. Yet a fourth strategy is to try both to prove one answer and to disprove it, and to use information gained in one of the processes to guide the other.

5.2 REPRESENTING INSTANCE AND *ISA* RELATIONSHIPS

In Chapter 4, we discussed the specific attributes *instance* and *isa* and described the important role they play in a particularly useful form of reasoning, property inheritance. But if we look back at the way we just represented our knowledge about Marcus and Caesar, we do not appear to have used these attributes at all. We certainly have not used predicates with those names. Why not? The answer is that although we have not used the predicates *instance* and *isa* explicitly, we have captured the relationships they are used to express, namely class membership and class inclusion.

Figure 5.3 shows the first five sentences of the last section represented in logic in three different ways. The first part of the figure contains the representations we have already discussed. In these representations, class membership is represented with unary predicates (such as *Roman*), each of which corresponds to a class. Asserting that $P(x)$ is true is equivalent to asserting that x is an instance (or element) of P. The second part of the figure contains representations that use the *instance* predicate explicitly. The predicate *instance* is a binary one, whose first argument is an object and whose second argument is a class to which the object belongs. But these representations do not use an explicit *isa* predicate. Instead, subclass relationships, such as that between Pompeians and Romans, are described as shown in sentence 3. The implication rule there states that if an object is an instance of the subclass *Pompeian* then it is an instance of the superclass *Roman*. Note that this rule is equivalent to the standard set-theoretic definition of the subclass-superclass relationship. The third part contains representations that use both the *instance* and *isa* predicates explicitly. The use of the *isa* predicate simplifies the representation of sentence 3, but it requires that one additional axiom (shown here as number 6) be provided. This additional axiom describes how an *instance* relation and *an isa* relation can be combined to derive a new *instance* relation. This one additional axiom is general, though, and does not need to be provided separately for additional *isa* relations.

These examples illustrate two points. The first is fairly specific. It is that, although class and superclass memberships are important facts that need to be represented, those memberships need not be represented with predicates labeled *instance* and *isa*. In fact, in a logical framework it is usually unwieldy to do that, and instead unary predicates corresponding to the classes are often used. The second point is more general. There are usually several different ways of representing a given fact within a particular representational framework, be it logic or anything else. The choice depends partly on which deductions need to be supported most efficiently and partly on taste. The only important thing is that within a particular knowledge base consistency of representation is critical. Since any particular inference rule is designed to work on one particular form of representation, it is necessary that all the knowledge to which that rule is intended to apply be in the form that the rule demands. Many errors in the reasoning performed by knowledge-based programs are the result of inconsistent representation decisions. The moral is simply to be careful.

There is one additional point that needs to be made here on the subject of the use of *isa* hierarchies in logic-based systems. The reason that these hierarchies are so important is not that they permit the inference of superclass membership. It is that by permitting the inference of superclass membership, they permit the inference of other properties associated with membership in that superclass. So, for example, in our sample knowledge base it is important to be able to conclude that Marcus is a Roman because we have some relevant knowledge about Romans, namely that they either hate Caesar or are loyal to him. But recall that in the baseball example of Chapter 4, we were able to associate knowledge with superclasses that could then be overridden by more specific knowledge associated either with individual instances or with subclasses. In other words, we recorded default values that could be accessed whenever necessary. For example, there was a height associated with adult males and a different height associated with baseball players. Our procedure for manipulating the *isa* hierarchy guaranteed that we always found the correct (i.e., most specific) value for any attribute. Unfortunately, reproducing this result in logic is difficult.

Suppose, for example, that, in addition to the facts we already have, we add the following.[1]

$$Pompeian(Paulus)$$
$$\neg\ [loyalto(Paulus,\ Caesar) \vee hate(Paulus, Caesar)]$$

In other words, suppose we want to make Paulus an exception to the general rule about Romans and their feelings toward Caesar. Unfortunately, we cannot simply add these facts to our existing knowledge base the way we could just add new nodes into a semantic net. The difficulty is that if the old assertions are left unchanged, then the addition of the new assertions makes the knowledge base inconsistent. In order to restore consistency, it is necessary to modify the original assertion to which an exception is being made. So our original sentence 5 must become:

$$\forall x : Roman(x) \wedge \neg eq(x, Paulus) \rightarrow loyalto(x, Caesar) \vee hate(x, Caesar)$$

In this framework, every exception to a general rule' must be stated twice, once in a particular statement and once in an exception list that forms part of the general rule. This makes the use of general rules in this framework less convenient and less efficient when there are exceptions than is the use of general rules in a semantic net.

A further problem arises when information is incomplete and it is not possible to prove that no exceptions apply in a particular instance. But we defer consideration of this problem until Chapter 7.

5.3 COMPUTABLE FUNCTIONS AND PREDICATES

In the example we explored in the last section, all the simple facts were expressed as combinations of individual predicates, such as:

tryassassinate(Marcus,Caesar)

This is fine if the number of facts is not very large or if the facts themselves are sufficiently unstructured that there is little alternative. But suppose we want to express simple facts, such as the following greater-than and less-than relationships:

gt(1,0)	lt(0,1)
gt(2,1)	lt(1,2)
gt(3,2)	lt(2,3)
⋮	⋮

Clearly we do not want to have to write out the representation of each of these facts individually. For one thing, there are infinitely many of them. But even if we only consider the finite number of them that can be represented, say, using a single machine word per number, it would be extremely inefficient to store explicitly a large set of statements when we could, instead, so easily compute each one as we need it. Thus it becomes useful to augment our representation by these *computable predicates*. Whatever proof procedure we use, when it comes upon one of these predicates, instead of searching for it explicitly in the database or attempting to deduce it by further reasoning, we can simply invoke a procedure, which we will specify in addition to our regular rules, that will evaluate it and return true or false.

It is often also useful to have computable functions as well as computable predicates. Thus we might want to be able to evaluate the truth of

gt(2 + 3,1)

To do so requires that we first compute the value of the plus function given the arguments 2 and 3, and then send the arguments 5 and 1 to *gt*.

The next example shows how these ideas of computable functions and predicates can be useful. It also makes use of the notion of equality and allows equal objects to be substituted for each other whenever it appears helpful to do so during a proof.

Consider the following set of facts, again involving Marcus:

1. Marcus was a man.

   $man(Marcus)$

   Again we ignore the issue of tense.
2. Marcus was a Pompeian.

   $Pompeian(Marcus)$

3. Marcus was born in 40 A.D.

   $born(Marcus, 40)$

   For simplicity, we will not represent A.D. explicitly, just as we normally omit it in everyday discussions. If we ever need to represent dates B.C., then we will have to decide on a way to do that, such as by using negative numbers. Notice that the representation of a sentence does not have to look like the sentence itself as long as there is a way to convert back and forth between them. This allows us to choose a representation, such as positive and negative numbers, that is easy for a program to work with.
4. All men are mortal.

   $\forall x: man(x) \rightarrow mortal(x)$

5. All Pompeians died when the volcano erupted in 79 A.D.

   $erupted(volcano, 79) \wedge \forall x : [Pompeian(x) \rightarrow died(x, 79)]$

   This sentence clearly asserts the two facts represented above. It may also assert another that we have not shown, namely that the eruption of the volcano caused the death of the Pompeians. People often assume causality between concurrent events if such causality seems plausible.

   Another problem that arises in interpreting this sentence is that of determining the referent of the phrase "the volcano." There is more than one volcano in the world. Clearly the one referred to here is Vesuvius, which is near Pompeii and erupted in 79 A.D. In general, resolving references such as these can require both a lot of reasoning and a lot of additional knowledge.
6. No mortal lives longer than 150 years.

   $\forall x : \forall t_1 : \forall t_2 : mortal(x) \wedge born(x, t_1) \wedge gt(t_2 - t_1, 150) \rightarrow dead(x, t_2)$

   There are several ways that the content of this sentence could be expressed. For example, we could introduce a function *age* and assert that its value is never greater than 150. The representation shown above is simpler, though, and it will suffice for this example.
7. It is now 1991.

   $now = 1991$

   Here we will exploit the idea of equal quantities that can be substituted for each other.

Now suppose we want to answer the question "Is Marcus alive?" A quick glance through the statements we have suggests that there may be two ways of deducing an answer. Either we can show that Marcus is dead because he was killed by the volcano or we can show that he must be dead because he would otherwise be more than 150 years old, which we know is not possible. As soon as we attempt to follow

either of those paths rigorously, however, we discover, just as we did in the last example, that we need some additional knowledge. For example, our statements talk about dying, but they say nothing that relates to being alive, which is what the question is asking. So we add the following facts:

8. Alive means not dead.

$$\forall x : \forall t : [alive(x, t) \rightarrow \neg dead(x,t)] \wedge [\neg dead(x, t) \rightarrow alive(x, t)]$$

This is not strictly correct, since $\neg dead$ implies alive only for animate objects. (Chairs can be neither dead nor alive.) Again, we will ignore, this for now. This is an example of the fact that rarely do two expressions have truly identical meanings in all circumstances.

9. If someone dies, then he is dead at all later times.

$$\forall x : \forall t_1 : \forall t_2 : died(x, t_1) \wedge gt(t_2, t_1) \rightarrow dead(x, t_2)$$

This representation says that one is dead in all years after the one in which one died. It ignores the question of whether one is dead in the year in which one died.

1. $man(Marcus)$
2. $Pompeian(Marcus)$
3. $born(Marcus, 40)$
4. $\forall x : man(x) \rightarrow mortal(x)$
5. $\forall : Pompeian(x) \rightarrow died(x, 79)$
6. $erupted(volcano, 79)$
7. $\forall_x : \forall t_1 : \forall t_2 : mortal(x) \wedge born(x, t_1) \wedge gt(t_2 - t_1, 150) \rightarrow dead(x, t_2)$
8. $now = 1991$
9. $\forall x : \forall t : [alive(x, t) \rightarrow \neg dead(x, t)] \wedge [\neg dead(x, t) \rightarrow alive(x, t)]$
10. $\forall x : \forall t_1 : \forall t_2 : died(x, t_1) \wedge gt(t_2, t_1) \rightarrow dead(x, t_2)$

Fig. 5.4 *A Set of Facts about Marcus*

To answer that requires breaking time up into smaller units than years. If we do that, we can then add rules that say such things as "One is dead at *time* (*year 1, month 1*) if one died during (*year 1, month 1*) and *month 2* precedes *month 1*]." We can extend this to days, hours, etc., as necessary. But we do not want to reduce all time statements to that level of detail, which is unnecessary and often not available.

A summary of all the facts we have now represented is given in Fig. 5.4. (The numbering is changed slightly because sentence 5 has been split into two parts.) Now let's attempt to answer the question "Is Marcus alive?" by proving:

$$\neg alive(Marcus, now)$$

Two such proofs are shown in Fig. 5.5 and 5.6. The term *nil* at the end of each proof indicates that the list of conditions remaining to be proved is empty and so the proof has succeeded. Notice in those proofs that whenever a statement of the form:

$$a \wedge b \rightarrow c$$

was used, a and b were set up as independent subgoals. In one sense they are, but in another sense they are not if they share the same bound variables, since, in that case, consistent substitutions must be made in each of them. For example, in Fig. 5.6 look at the step justified by statement 3. We can satisfy the goal

$born(Marcus, t_1)$

using statement 3 by binding A to 40, but then we must also bind A to 40 in

$gt(now - t_1, 150)$

since the two t_1's were the same variable in statement 4, from which the two goals came. A good computational proof procedure has to include both a way of determining that a match exists and a way of guaranteeing uniform substitutions throughout a proof. Mechanisms for doing both those things are discussed below.

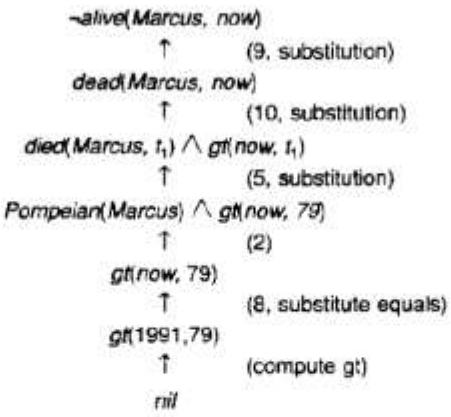


Fig. 5.5 *One Way of Proving That Marcus Is Dead*

From looking at the proofs we have just shown, two things should be clear:

- Even very simple conclusions can require many steps to prove.
- A variety of processes, such as matching, substitution, and application of *modus ponens* are involved in the production of a proof. This is true even for the simple statements we are using. It would be worse if we had implications with more than a single term on the right or with complicated expressions involving *ands* and *ors* on the left.

The first of these observations suggests that if we want to be able to do nontrivial reasoning, we are going to need some statements that allow us to take bigger steps along the way. These should represent the facts that people gradually acquire as they become experts. How to get computers to acquire them is a hard problem for which no very good answer is known.

The second observation suggests that actually building a program to do what people do in producing proofs such as these may not be easy. In the next section, we introduce a proof procedure called *resolution* that reduces some of the complexity because it operates on statements that have first been converted to a single canonical form.

5.4.3 Resolution in Propositional Logic

In order to make it clear how resolution works, we first present the resolution procedure for propositional logic. We then expand it to include predicate logic.

In propositional logic, the procedure for producing a proof by resolution of proposition P with respect to a set of axioms F is the following.

Algorithm: Propositional Resolution

1. Convert all the propositions of F to clause form.
2. Negate P and convert the result to clause form. Add it to the set of clauses obtained in step 1.
3. Repeat until either a contradiction is found or no progress can be made:
   (a) Select two clauses. Call these the parent clauses.
   (b) Resolve them together. The resulting clause, called the *resolvent*, will be the disjunction of all of the literals of both of the parent clauses with the following exception: If there are any pairs of literals L and $\neg L$ such that one of the parent clauses contains L and the other contains $\neg L$, then select one such pair and eliminate both L and $\neg L$ from the resolvent.
   (c) If the resolvent is the empty clause, then a contradiction has been found. If it is not, then add it to the set of clauses available to the procedure.

Let's look at a simple example. Suppose we are given the axioms shown in the first column of Fig. 5.7 and we want to prove R. First we convert the axioms to clause form, as shown in the second column of the figure.

Given Axioms	Converted to Clause Form	
P	P	(1)
$(P \wedge Q) \rightarrow R$	$\neg P \vee \neg Q \vee R$	(2)
$(S \vee T) \rightarrow Q$	$\neg S \vee Q$	(3)
	$\neg T \vee Q$	(4)
T	T	(5)

Fig. 5.7 *A Few Facts in Propositional Logic*

Then we negate R, producing $\neg R$, which is already in clause form. Then we begin selecting pairs of clauses to resolve together. Although any pair of clauses can be resolved, only those pairs that contain complementary literals will produce a resolvent that is likely to lead to the goal of producing the empty clause (shown as a box). We might, for example, generate the sequence of resolvents shown in Fig. 5.8. We begin by resolving with the clause $\neg R$ since that is one of the clauses that must be involved in the contradiction we are trying to find.

One way of viewing the resolution process is that it takes a set of clauses that are all assumed to be true and, based on information provided by the others, it generates new clauses that represent restrictions on the way each of those original clauses can be made true. A contradiction occurs when a clause becomes so restricted that there is no way it can be true. This is indicated by the generation of the empty clause. To see how this works, let's look again at the example. In order for proposition 2 to be true., one of three things must be true: $\neg P$, $\neg Q$, or R. But we are assuming that $\neg R$ is true. Given that, the only way for proposition 2 to be true is for one of two things to be true: $\neg P$ or $\neg Q$. That is what the first resolvent clause says. But proposition 1 says that P is true, which means that $\neg P$ cannot be true, which leaves only one way for proposition 2 to be true, namely for $\neg Q$ to be true (as shown in the second resolvent clause). Proposition 4 can be true if either $\neg T$ or Q is true. But since we now know that $\neg Q$ must be true, the only way for proposition 4 to be true is for $\neg T$ to be true (the third resolvent). But proposition 5 says that T is true. Thus there is no way for all of these clauses to be true in a single interpretation. This is indicated by the empty clause (the last resolvent).

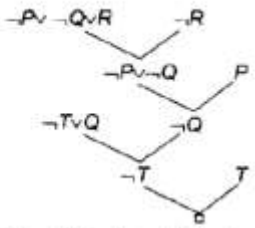


Fig. 5.8 *Resolution in Propositional Logic*

5.4.5 Resolution in Predicate Logic

We now have an easy way of determining that two literals are contradictory—they are if one of them can be unified with the negation of the other. So, for example, $man(x)$ and $\neg man(Spot)$ are contradictory, since $man(x)$ and $man(Spot)$ can be unified. This corresponds to the intuition that says that $man(x)$ cannot be true for all x if there is known to be some x, say Spot, for which $man(x)$ is false. Thus in order to use resolution for expressions in the predicate logic, we use the unification algorithm to locate pairs of literals that cancel out.

We also need to use the unifier produced by the unification algorithm to generate the resolvent clause. For example, suppose we want to resolve two clauses:

1. $man(Marcus)$
2. $\neg man(x_1) \vee mortal(x_1)$

The literal $man(Marcus)$ can be unified with the literal $man(x)$ with the substitution $Marcus/x_1$, telling us that for $x_1 = Marcus$, $\neg man(Marcus)$ is false. But we cannot simply cancel out the two *man* literals as we did in propositional logic and generate the resolvent $mortal(x_1)$. Clause 2 says that for a given x_1, either $\neg man(x_1)$ or $mortal(x_1)$. So for it to be true, we can now conclude only that $mortal(Marcus)$ must be true. It is not necessary that $mortal(x_1)$ be true for all x_1, since for some values of x_1, $\neg man(x_1)$ might be true, making $mortal(x_1)$ irrelevant to the truth of the complete clause. So the resolvent generated by clauses 1 and 2 must be $mortal(Marcus)$, which we get by applying the result of the unification process to the resolvent. The resolution process can then proceed from there to discover whether $mortal(Marcus)$ leads to a contradiction with other available clauses.

This example illustrates the importance of standardizing variables apart during the process of converting expressions to clause form. Given that that standardization has bee'n done, it is easy to determine how the

unifier must be used to perform substitutions to create the resolvent. If two instances of the same variable occur, then they must be given identical substitutions.

We can now state the resolution algorithm for predicate logic as follows, assuming a set of given statements F and a statement to be proved P:

Algorithm: Resolution

1. Convert all the statements of F to clause form.
2. Negate P and convert the result to clause form. Add it to the set of clauses obtfHied in 1.
3. Repeat until either a contradiction is found, no progress can be made, or a prede- termined amount of effort has been expended.
   (a) Select two clauses. Call these the parent clauses.
   (b) Resolve them together. The resolvent will be the disjunction of all the literals of both parent clauses with appropriate substitutions performed and with the following exception: If there is one pair of literals $T1$ and $\neg T2$ such that one of the parent clauses contains $T2$ and the other contains $T1$ and if $T1$ and $T2$ are unifiable, then neither $T1$ nor $T2$ should appear in the resolvent. We call $T1$ and $T2$ *Complementary literals.* Use the substitution produced by the unification to create the resolvent. If there is more than one pair of complementary literals, only one pair should be omitted from the resolvent.
   (c) If the resolvent is the empty clause, then a contradiction has been found. If it is not, then add it to the set of clauses available to the procedure.

If the choice of clauses to resolve together at each step is made in certain systematic ways, then the resolution procedure will find a contradiction if one exists. However, it may take a very long time. There exist strategies for making the choice that can speed up the process considerably:

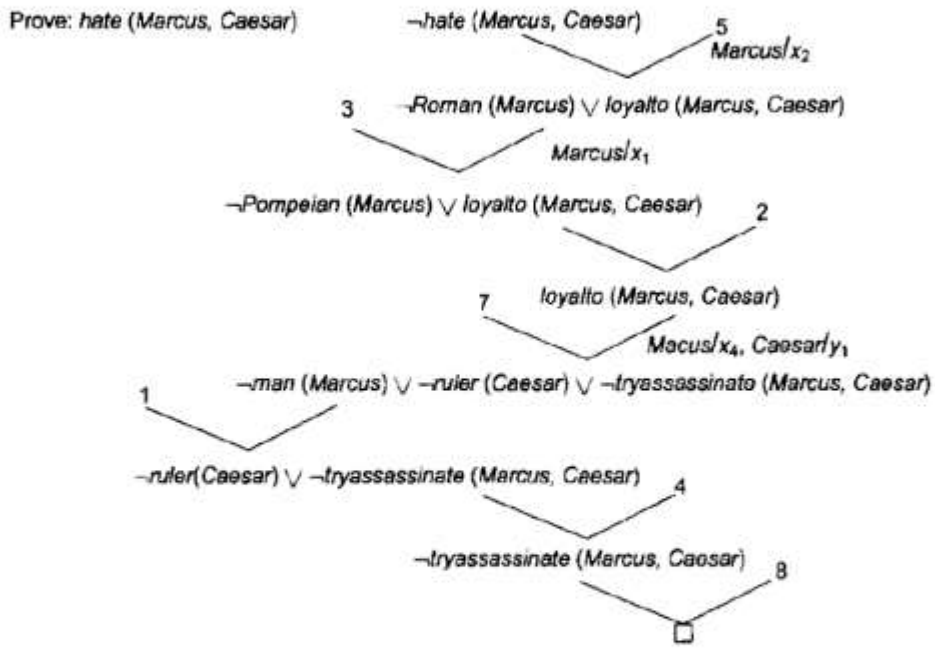
- Only resolve pairs of clauses that contain complementary literals, since only such resolutions produce new clauses that are harder to satisfy than their parents. To facilitate this, index clauses by the predicates they contain, combined with an indication of whether the predicate is negated. Then, given a particular clause, possible resolvents that contain a complementary occurrence of one of its predi- cates can be located directly.
- Eliminate certain clauses as soon as they are generated so that they cannot partic- ipate in later resolutions. Two kinds of clauses should be eliminated: tautologies (which can never be unsatisfied) and clauses that are subsumed by other clauses (i.e., they are easier ;o satisfy. For example, $P \vee Q$ is subsumed by P.)
- Whenever possible, resolve either with one of the clauses that is part of the statement we are trying to refute or with a clause generated by a resolution with such a clause. This is called the *set-of-support strategy* and corresponds to the intuition that the contradiction we are looking for must involve the statement we are trying to prove. Any other contradiction would say that the previously believed statements were inconsistent.
- Whenever possible, resolve with clauses that have a single literal. Such resolutions generate new clauses with fewer literals than the larger of their parent clauses and thus are probably closer to the goal of a resolvent with zero terms. This method is called the *unit-preference strategy.*

Let's now return to our discussion of Marcus and show how resolution can be used to prove new things about him. Let's first consider the set of statements introduced in Section 5.1. To use them in resolution proofs, we must convert them to clause form as described in Section 5.4.1. Figure 5.9(a) shows the results of that conversion. Figure 5.9(b) shows a resolution proof of the statement

Axioms in clause form:

1. $man(Marcus)$
2. $Pompeian(Marcus)$
3. $\neg Pompeian(x_1) \vee Roman(x_1)$
4. $ruler(Caesar)$
5. $\neg Roman(x_2) \vee loyalto(x_2, Caesar) \vee hate(x_2, Caesar)$
6. $loyalto(x_3, f(x_3))$
7. $\neg man(x_4) \vee \neg ruler(y_1) \vee \neg tryassassinate(x_4, y_1) \vee loyalto(x_4, y_1)$
8. $tryassassinate(Marcus, Caesar)$

(a)



(b)

Fig. 5.9 *A Resolution Proof*

Of course, many more resolvents could have been generated than we have shown, but we used the heuristics described above to guide the search. Notice that what we have done here essentially is to reason backward from the statement we want to show is a contradiction through a set of intermediate conclusions to the final conclusion of inconsistency.

Suppose our actual goal in proving the assertion

$$hate(Marcus, Caesar)$$

was to answer the question "Did Marcus hate Caesar?" In that case, we might just as easily have attempted to prove the statement

$$\neg hate(Marcus, Caesar)$$

To do so, we would have added

hate(Marcus, Caesar)

to the set of available clauses and begun the resolution process. But immediately we notice that there are no clauses that contain a literal involving $\neg hate$. Since the resolution process can only generate new clauses that are composed of combinations of literals from already existing clauses, we know that no such clause can be generated and thus we conclude that *hate(Marcus,Caesar)* will not produce a contradiction with the known statements. This is an example of the kind of situation in which the resolution procedure can detect that no contradiction exists. Sometimes this situation is detected not at the beginning of a proof, but part way through, as shown in the example in Figure 5.10(*a*), based on the axioms given in Fig. 5.9.

But suppose our knowledge base contained the two additional statements

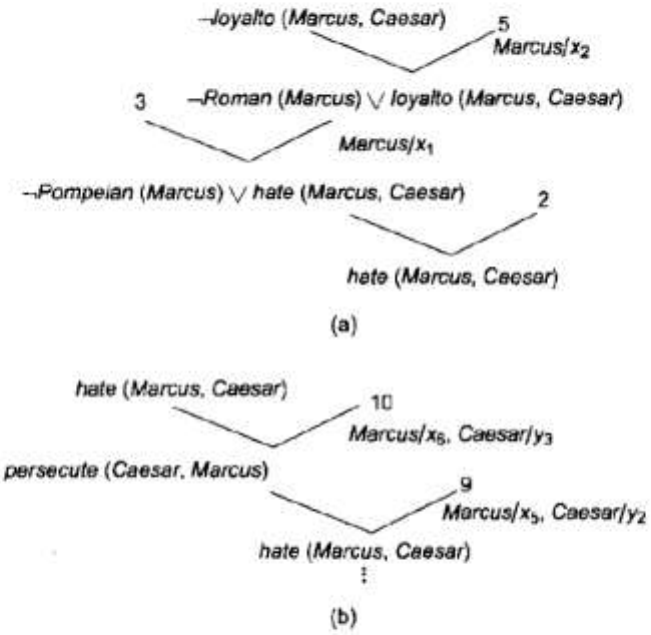


Fig. 5.10 *An Unsuccessful Attempt at Resolution*

9. $persecute(x, y) \rightarrow hate(y, x)$
10. $hate(x, y) \rightarrow persecute(y, x)$

Converting to clause form, we get

9. $\neg persecute(x_5, y_2) \vee hate(y_2, x_5)$
10. $\neg hate(x_6, y_3) \vee persecute(y_3, x_6)$

These statements enable the proof of Fig. 5.10(*a*) to continue as shown in Fig. 5.10(*b*). Now to detect that there is no contradiction we must discover that the only resolvents that can be generated have been generated before. In other words, although we can generate resolvents, we can generate no new ones.

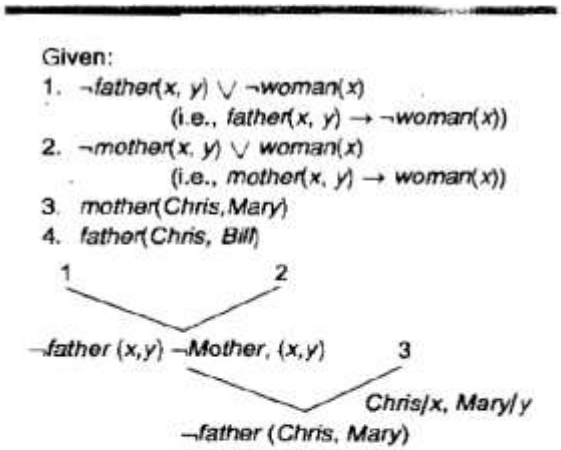


Fig. 5.11 *The Need to Standardize Variables*

Axioms in clause form:

1. $man(Marcus)$
2. $Pompeian(Marcus)$
3. $born(Marcus, 40)$
4. $\neg man(x_1) \vee mortal(x_1)$
5. $\neg Pompeian(x_2) \vee died(x_2, 79)$
6. $erupted(volcano, 79)$
7. $\neg morta(x_3) \vee \neg born(x_3, t_1) \vee \neg gt(t_2 - t_1, 150) \vee dead(x_3, t_2)$
8. $now = 2008$

9a. $\neg alive(x_4, t_3) \vee \neg dead(x_4, t_3)$

9b. $dead(x_5, t_4) \vee alive(x_5, t_4)$

10. $\neg died(x_6, t_5) \vee \neg gt(t_6, t_5) \vee dead(x_6, t_6)$

Prove: $\neg alive(Marcus, now)$

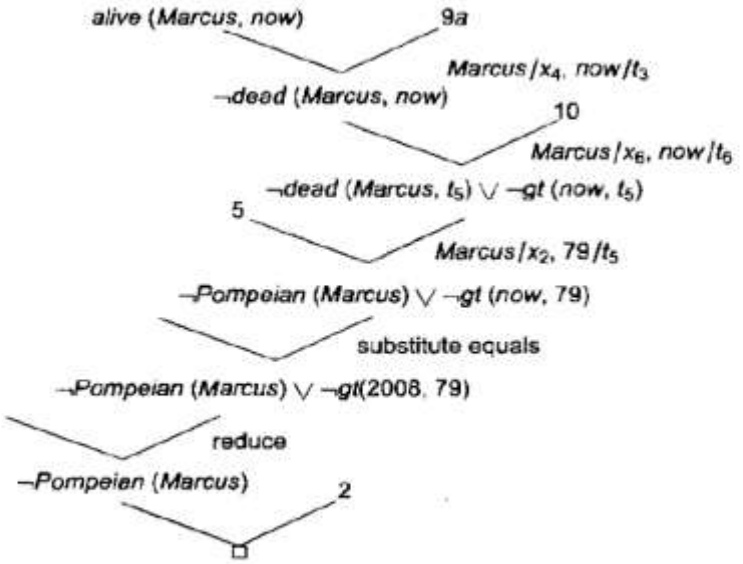


Fig. 5.12 *Using Resolution with Equality and Reduce*

Recall that the final step of the process of converting a set of formulas to clause form was to standardize apart the variables that appear in the final clauses. Now that we have discussed the resolution procedure, we can see clearly why this step is so important. Figure 5.11 shows an example of the difficulty that may arise if standardization is not done. Because the variable *y* occurs in both clause 1 and clause 2, the substitution at the second resolution step produces a clause that is too restricted and so does not lead to the contradiction that is present in the database. If, instead, the clause

$$\neg father(Chris, y)$$

had been produced, the contradiction with clause 4 would have emerged. This would have happened if clause 2 had been rewritten as

$$\neg mother(a, b) \vee woman(a)$$

In its pure form, resolution requires all the knowledge it uses to be represented in the form of clauses. But as we pointed out in Section 5.3, it is often more efficient to represent certain kinds of information in the form of computable functions, computable predicates, and equality relationships. It is not hard to augment resolution to handle this sort of knowledge. Figure 5.12 shows, a resolution proof of the statement

$$\neg alive(Marcus,now)$$

based on the statements given in Section 5.3. We have added two ways of generating new clauses, in addition to the resolution rule:

- Substitution of one value for another to which it is equal.
- Reduction of computable predicates. If the predicate evaluates to FALSE, it can simply be dropped, since adding V FALSE to a disjunction cannot change its truth value. If the predicate evaluates to TRUE, then the generated clause is a tautology and cannot lead to a contradiction.

Unification

A substitution S is a finite (possibly empty) set of pairs $X_i \leftarrow t_i$ where X_i is a variable and t_i is a term; $X_i \neq X_j$ for $i \neq j$; and X_i does not occur in t_j for any i, j. The notation TS indicates the application of a substitution to a term T; each occurrence in T of a variable in S is replaced by its corresponding term. We can also apply one substitution to another substitution (provided the two substitutions do not have variables in common), replacing variables in the terms on the right hand sides of the pairs.

A unifier of two terms is a substitution that makes the terms identical. The resulting term is a common instance of the two terms.

A most general unifier of two terms is a unifier that produces a most general common instance. If two terms unify, there is a most general unifier.

For example

man(john) and ¬ man(john) is a contradiction. ,but man(john) and ¬ man(spot) is not a contradiction..thus inorder to determine contradictions,we need a matching procedure that compares 2 literals and discover whether there exists a set of substitutions that makes them identical. This is known as unification algorithm

algorithm: Unify(L1,L2)

If L1 and L2 are both variables or constants then

a) if L1 and L2 are identical,then return NIL.
b) Else if L1 is a variable,then if L1 occurs in L2 then return {FAIL],else return (L2/L1)
c) Else if L2 is a variable then if L2 occurs in L1 then return {FAIL].,else return(L1/L2)
d) Else return{FAIL].

2. If the initial predicate symbols in L1 and L2 are not identical then return{FAIL]

3. If L1 and L2 have a different number of arguments,then return {FAIL}

4. Set SUBST to NIL.

5. for i← 1 to number of arguments in L1:

a)call unify with the ith argument of L1 and the Ith argument of L2,putting result in S.

b)If S contains FAIL ,then return{FAIL}

c)If S is not equal to NIL then:

i)Apply S to the remainder of both L1 and L2.

ii)SUBST :=APPEND(S,SUBST)

6. return SUBST.

Resolution

Resolution is a method of proving statements. Resolution produces proofs by refutation. In other words ,to prove a statement ,resolution attempts to show that the negation of the statement produces a contradiction of the known statements. For resolution the statements should be converted to the standard clausal form.

Conversions to CNF

1. Eliminate implications, equivalences

$$(p \Rightarrow q) \rightarrow (\neg p \vee q)$$

2. Move negations inside (DeMorgan's Laws, double negation)

$$\neg(p \wedge q) \rightarrow \neg p \vee \neg q$$
$$\neg(p \vee q) \rightarrow \neg p \wedge \neg q$$
$$\neg \forall x\ p \rightarrow \exists x\ \neg p$$
$$\neg \exists x\ p \rightarrow \forall x\ \neg p$$
$$\neg\neg\ p \rightarrow p$$

3. Standardize variables (rename duplicate variables)

$$(\forall x\ P(x)) \vee (\exists x\ Q(x)) \rightarrow (\forall x\ P(x)) \vee (\exists y\ Q(y))$$

4. Move all quantifiers left (no invalid capture possible)

$$(\forall x\ P(x)) \vee (\exists y\ Q(y)) \rightarrow \forall x\ \exists y\ P(x) \vee Q(y)$$

5. Skolemization (removal of existential quantifiers through elimination)

- If no universal quantifier occurs before the existential quantifier, replace the variable with a new constant symbol

  $$\exists y\ P(A) \vee Q(y) \rightarrow P(A) \vee Q(B)$$

- If a universal quantifier precede the existential quantifier replace the variable with a function of the "universal" variable

  $$\forall x\ \exists y\ P(x) \vee Q(y) \rightarrow \forall x\ \ P(x) \vee Q(F(x))$$

  $F(x)$ - **a Skolem function**

6. Drop universal quantifiers (all variables are universally quantified)

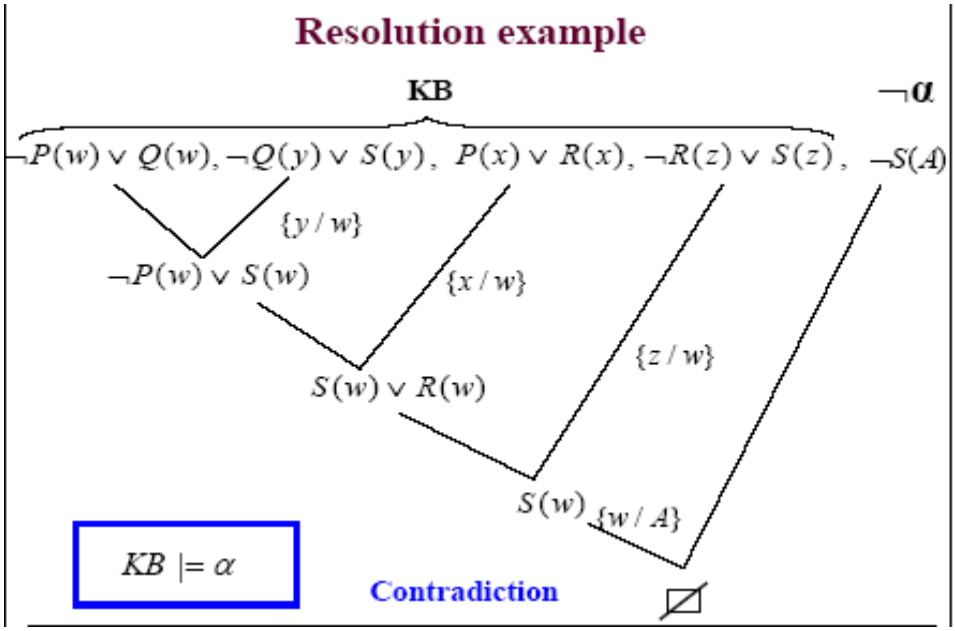
$$\forall x \;\; P(x) \vee Q(F(x)) \rightarrow P(x) \vee Q(F(x))$$

7. Convert to CNF using the distributive laws

$$p \vee (q \wedge r) \rightarrow (p \vee q) \wedge (p \vee r)$$

The result is a CNF with variables, constants, functions

Resolution example



Sentences in Horn normal form

- **Horn normal form (HNF) in the propositional logic**
  - a special type of clause with **at most one positive literal**

$$(A \vee \neg B) \wedge (\neg A \vee \neg C \vee D) \wedge A$$

Can be written as:

$$(B \Rightarrow A) \wedge ((A \wedge C) \Rightarrow D) \wedge A$$

- A clause with one literal, is also called **a fact**

  Example: A **is a fact**
- A clause representing an implication (with a conjunction of positive literals in antecedent and one positive literal in consequent), is also called **a rule**

  Example: $(A \wedge C) \Rightarrow D$ **is a rule**

Sentences in Horn normal form

- **Horn normal form (HNF) in the propositional logic**
  - a special type of clause with **at most one positive literal**

$$(A \vee \neg B) \wedge (\neg A \vee \neg C \vee D) \wedge A$$

Can be written as:

$$(B \Rightarrow A) \wedge ((A \wedge C) \Rightarrow D) \wedge A$$

- **Modus ponens:** $\dfrac{A \Rightarrow B, \quad A}{B}$
  - is the **complete inference rule** for KBs in the Horn normal form.

First-order logic (FOL)

- **adds variables and quantifiers, works with terms**

Generalized modus ponens rule:

$$\sigma = \text{a substitution s.t. } \forall i \; SUBST(\sigma, \phi_i') = SUBST(\sigma, \phi_i)$$

$$\frac{\phi_1', \phi_2', \ldots, \phi_n', \quad \phi_1 \wedge \phi_2 \wedge \ldots \phi_n \Rightarrow \tau}{SUBST(\sigma, \tau)}$$

Generalized modus ponens:

- is **complete** for the KBs with sentences in the Horn form;
- Not all first-order logic sentences can be expressed in this form

Algorithm :Resolution

1.Convert all statements of F in to clausal form.
2.Negate P and convert the result to clausal form. Add it to the set of clauses obtained in step 1.
3.Repeat until either a contradiction is found or no progress can be made.

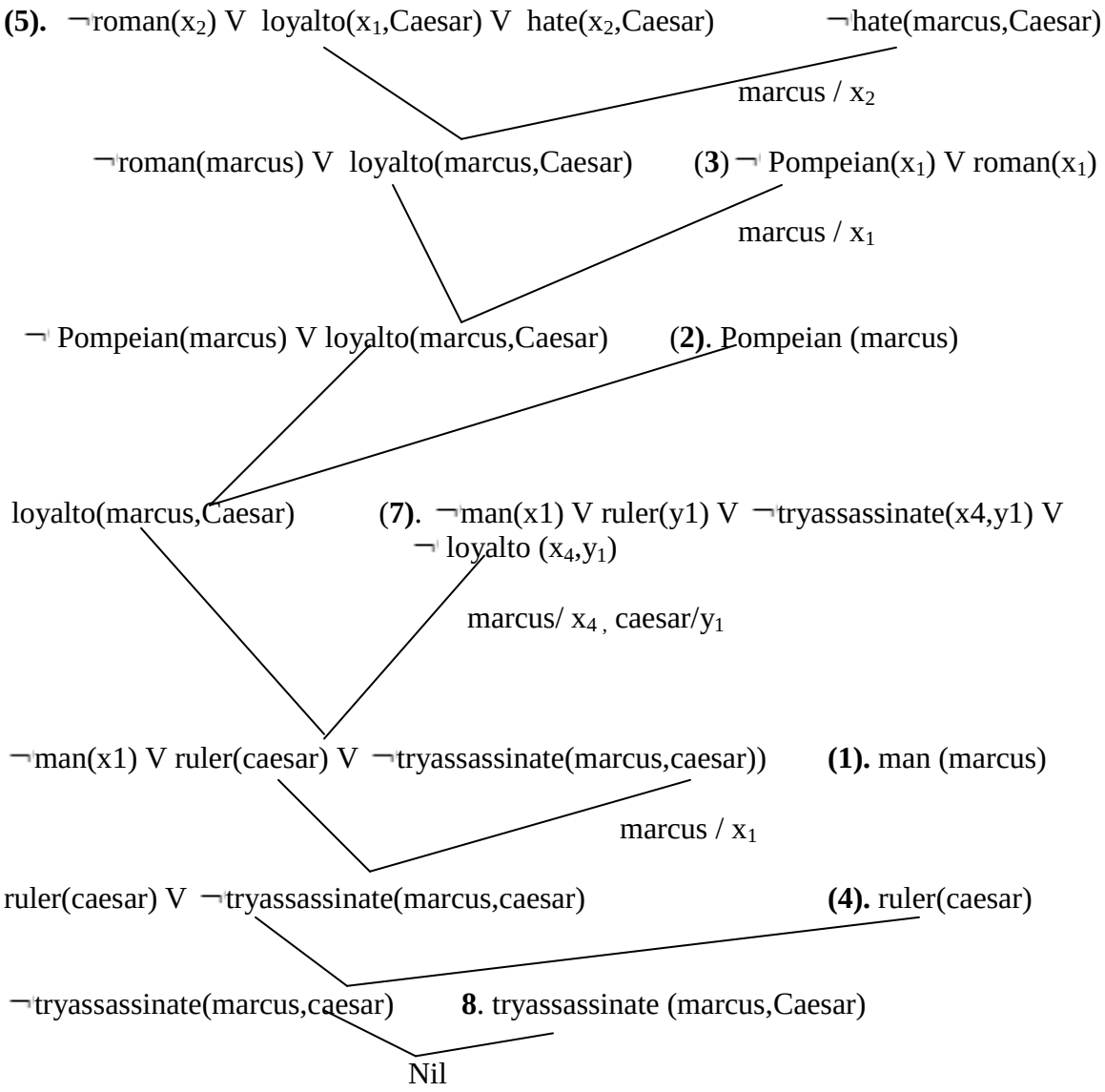
a) Select 2 clauses. Call these the parent clauses.

b) Resolve them together. If there is one pair of literals T1 and –T2 such that one pair of parent clauses contains T1 and the other contains T2and if T1 and T2 are unifiable,then neither T1 nor T2 should appear in the resolvent.

c)If the resolvent is the mpty clause,then a contradiction hes been found. If it is not then add it to the set of clauses available to the procedure.

Consider the following sentences in clausal form
1. man (marcus)
2. Pompeian (marcus)
3. ¬ Pompeian(x_1) V roman(x_1)
4. ruler(Caesar)
5. ¬roman(x_2) V loyalto(x_1,Caesar) V hate(x_2,Caesar)
6. loyalto(x_3,fl(x_3))
7. ¬man(x1) V ruler(y1) V ¬tryassassinate(x4,y1) V ¬ loyalto (x4,y1)
8. tryassassinate (marcus,Caesar)

Prove hate(marcus,Caesar)
Negate: ¬hate(marcus,Caesar)



Since the result is nil the statement is proved.

Forward and backward chaining

Forward chaining

Forward chaining is a means of utilizing a set of condition-action rules. In this mode of operation, a rule-based system is data-driven. We use the data to decide which rules can fire, then we fire one of those rules, which may add to the data in working memory and then we repeat this process until we (hopefully) establish a conclusion.

Reason forward from initial states

Begin building a tree of move sequences that might be solutions by starting with the initial configurations at the root of the tree. Generate 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 .Continue this until a configuration that matches the goal state is generated.

Forward-chaining is to be contrasted with backward chaining.

Backward chaining

Backward chaining is a means of utilizing a set of condition-action rules. In backward chaining, we work back from possible conclusions of the system to the evidence, using the rules backwards. Thus backward chaining behaves in a goal-driven manner.

One needs to know which possible conclusions of the system one wishes to test for. Suppose, for example, in a medical diagnosis expert system, that one wished to know if the data on the patient supported the conclusion that the patient had some particular disease, D.

In backward-chaining, the goal (initially) is to find evidence for disease D. To achieve this, one would search for all rules whose action-part included a conclusion that the patient had disease D. One would then take each such rule and examine, in turn, the condition part of the rule. To support the disease D hypothesis, one has to show that these conditions are true. Thus these conditions now become the goals of the backward-chaining production system. If the conditions are not supported directly by the contents of working memory, we need to find rules whose action-parts include these conditions as their conclusions. And so on, until either we have established a chain of reasoning

demonstrating that the patient has disease D, or until we can find no more rules whose action-parts include conditions that are now among our list of goals.

Reason backward from the goal state.

Begin building a tree of move sequences that might be solutions by starting with the goal configurations at the root of the tree. Generate next level of the tree by finding all the rules whose right sides match the root nodee and using their leftt sides to create the new configuratuions.Continue this until a configuration that matches the initial state is generated.

Backward-chaining is to be contrasted with forward chaining.