Unit-3 Knowledge Reasoning Planning.

Knowledge Representation

Representation and Mapping

In order to solve the complex problems encountered in artificial intelligence, one needs both a large amount of knowledge and some mechanisms for manipulating that knowledge to create solutions to new problems. So far we have concentrated on some very general methods of manipulating knowledge using search. These methods are sufficiently general that we have been able to discuss them without reference to the way the knowledge they need is represented. For example, in discussing the A* algorithm, we hid all the references to domain-specific knowledge in the generation of successors and the computation of the h' function.

Although these methods are useful and will form the skeleton of many of the methods we are about to discuss, their strength is limited precisely because of their generality. As we look in more detail at ways of representing knowledge, it becomes clear that specific knowledge representation models allow for more specific, more powerful inference mechanisms that operate on them. A variety of ways of representing knowledge (facts) have been exploited in A.I. programs. But before we can talk about them individually, we must consider the following point that pertains to all discussions of representation, namely that we are dealing with two different kinds of entities:

- Facts: truths in some relevant world. These are the things we want to represent.
- Representations of facts in some chosen formalism. These are the things we will actually be able to manipulate.

In order for the representations to be of any interest with respect to the world, there must also be functions that map from facts to representations and from representations back to facts.

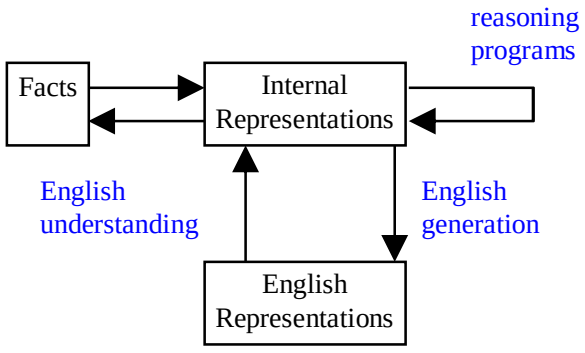


Figure 1: Mapping between Facts and Representations

One way to think of structuring these entities is as 2 levels:

- The knowledge level at which facts are described.

- The symbol level at which representations of objects at the knowledge level are defined in terms of symbols that can be manipulated by programs.

One representation of facts is so common that it deserves special mention: natural language sentences. Regardless of the representation for facts that we use in a program, we may also need to be concerned with a natural language representation of those facts in order to facilitate getting information into and out of the system. In this case, we must also have mapping functions from sentences to the representation we are actually going to use and from it back to sentences.

Let's look at a simple example using mathematical logic as the representational formalism. Consider the English sentence:

Spot is a dog.

The fact represented by that sentence can be represented in logic as

Dog(Spot)

Consider the English sentence:

Every dog has a tail

can be represented in logic as

$\forall x: dog(x) \rightarrow hastail(x)$

Suppose that we also have a representation of the fact that all dogs have tails. Then, using the deductive mechanisms of logic, we may generate the new representation object

Hastail(Spot)

Using an appropriate mapping function, we could then generate the English sentence

Spot has a tail.

Or we could make use of this representation of a new fact to cause us to take some appropriate action. Figure 1 shows how these three kinds of objects relate to each other.

It is important to keep in mind that usually the available mapping functions are not one-to-one. In fact, they are often not even functions, but rather many-to-many relations. (In other words, each object in the domain may map to several elements in the range, and several elements in the domain may map to the same element of the range.) This is particularly true of the mappings involving English representations of facts. For example, the two sentences "All dogs have tails" and "Every dog has a tail" could both represent the same fact. On the other hand, the former could represent either the fact that

every dog has at least one tail or the fact that each dog has several tails. As we will see shortly, when we try to convert English sentences into some other representation, such as logical propositions, we must first decide what facts the sentences represent and then convert those facts into the new representation.

Figure 2 shows the expanded view of figure 1. The dotted line across the top represents the abstract reasoning process that a program is intended to model. The solid line across the bottom represents the concrete reasoning process that a particular program performs. This program successfully models the abstract process to the extent that when the backward representation mapping is applied to the program's output the appropriate final facts are actually generated. If no good mapping can be defined for a problem then no matter how good the program to solve the problem, it will not be able to produce answers that correspond to real answers to the problem.

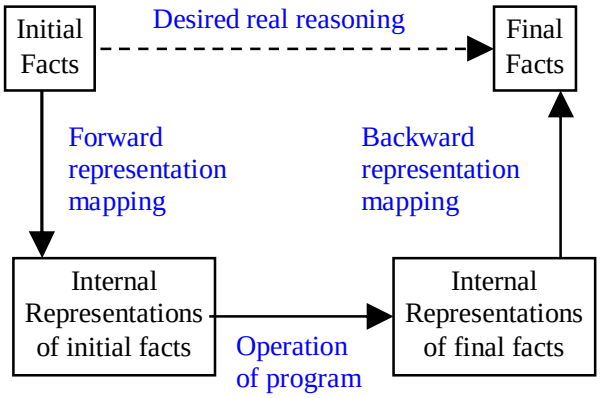


Figure 2: Representation of Facts

Approaches to Knowledge Representation

A good system for the representation of knowledge in a particular domain should possess the following four properties:

1. Representational Adequacy—the ability to represent all of the kinds of knowledge that are needed in that domain.
2. Inferential Adequacy—the ability to manipulate the representational structures in such a way as to derive new structures corresponding to new knowledge inferred from old.
3. Inferential Efficiency—the ability to incorporate into the knowledge structure additional information that can be used to focus the attention of the inference mechanisms in the most promising directions.
4. Acquisitional Efficiency-—the ability to acquire new information easily. The simplest case involves direct insertion, by a person, of new knowledge into the database. Ideally, the program itself would be able to control knowledge acquisition.

Unfortunately, no single system that optimizes all of the capabilities for all kinds of knowledge has yet been found. As a result, multiple techniques for knowledge representation exist. Many programs rely on more than one technique. In the chapters that

follow, the most important of these techniques are described in detail. But in this section, we provide a simple, example-based introduction to the important ideas.

1. **Simple Relational Knowledge**

The simplest way to represent declarative facts **is as a set of relations of the same sort used in database systems.** Figure 3 shows an example of such a relational system.

Player	Height	Weight	Bats-Throws
Hank Aaron	6-0	180	Right-Right
Willie Mays	5-10	170	Right-Right
Babe Ruth	6-2	215	Left-Left
Ted Williams	6-3	205	Left-Right

Figure 3: Simple Relational Knowledge

The reason that this representation is simple is that standing alone it provides very weak inferential capabilities. But knowledge represented in this form may serve as the input to more powerful inference engines. For example, given just the facts of Figure 3, it is not possible even to answer the simple question, "Who is the heaviest player?" But if a procedure for finding the heaviest player is provided, then these facts will enable the **procedure to compute an answer**. If, instead, we are provided with **a set of rules** for deciding which hitter to put up against a given pitcher (based on right and left handedness, say); then this same relation can provide at least some of the information required by those rules. Providing support for relational knowledge is what database systems are designed to do.

2. Inheritable Knowledge

The relational knowledge of Figure 3 corresponds to a set of attributes and associated values that together describe the objects of the knowledge base. Knowledge about objects, their attributes, and their values need not be as simple as that shown in our example. In particular, it is possible to augment the basic representation with **inference mechanisms** that operate on the structure of the representation. For this to be effective, the structure must be designed to correspond to the inference mechanisms that are desired. One of the most useful forms of inference is ***property inheritance***, in which **elements of specific classes inherit attributes** and values from more **general classes** in which they are included.

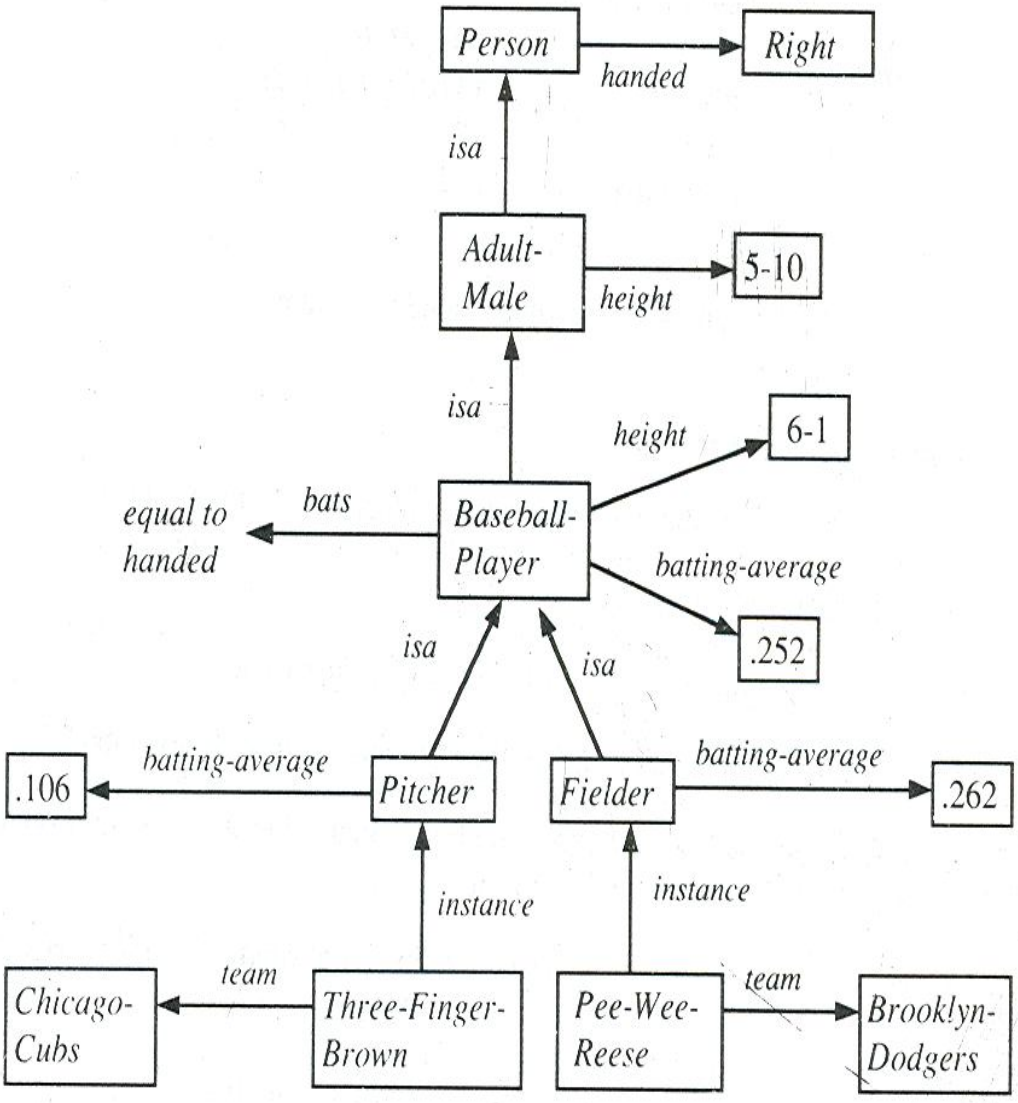


Figure 4: Inheritable Knowledge

In order to support property inheritance, objects must be organized into classes and classes must be arranged in a **generalization hierarchy**. Figure 4 shows some additional baseball knowledge inserted into a structure that is so arranged. Lines represent attributes. **Boxed nodes** represent **objects** and **values** of **attributes of objects**. These values can also be viewed as objects with attributes and values, and so on. The arrows on the lines point from an object to its value along the corresponding attribute line. The structure shown in the figure is a ***slot-and-filler structure***. It may also be called **a *semantic network*** or a **collection of *frames***. In the latter case: each individual **frame** represents the **collection** of **attributes** and **values** associated with a **particular node**. Figure 5 shows the node for baseball player displayed as a frame.

Baseball-Player

isa:	Adult-Male
Bats:	(EQUAL handed)
Height:	6-1
Batting average:	.252

Figure 5: Viewing a Node as a Frame

Usually the use of the term *frame system* implies somewhat more structure on the attributes and, the inference mechanisms that are available to apply to them than does the term ***semantic network.***

All of the objects and most of the attributes shown in this example have been chosen to correspond to the baseball domain, and they have no general significance. The two exceptions to this are the attribute ***isa***, which is being used to show class inclusion, and the attribute ***instance***, which is being used to show class membership. These two specific attributes provide the basis for property inheritance as an inference technique. Using this technique, the knowledge base can support retrieval both of facts that have been explicitly stored and of facts that can be derived from those that are explicitly stored.

An idealized form of the property inheritance algorithm can be stated as follows.

Algorithm: Property Inheritance

To retrieve a value *V* for attribute *A* of an instance object *O:*

1) Find *O* in the knowledge base.
2) If there is a value there for the attributed, report that value.
3) Otherwise, see if there is a value for the attribute *instance.* If not, then fail.
4) Otherwise, move to the node corresponding to that value and look for a value for the attribute *A.* If one is found, report it.
5) Otherwise, do until there is no value for the *isa* attribute or until an answer is found:
    a) Get the value of the *isa* attribute and move to that node.
    b) See if there is a value for the attributed. If there is, report it.

This procedure is simplistic. It does not say what we should do if there is more than one value of the *instance* or is*a* attribute. But it does describe the basic mechanism of inheritance. We can apply this procedure to our example knowledge base to derive answers to the following queries:
1. team(Pee-Wee-Reese) = Brooklyn-Dodgers. This attribute had a value stored explicitly in the knowledge base.
2. batting-average(Three-Finger-Brown) = .106. Since there is no value for batting average stored explicitly for Three Finger Brown, we follow the *instance* attribute to *Pitcher* and extract the value stored there. Now we observe one of the critical characteristics of property inheritance, namely that it may produce default values that are not guaranteed to be correct but that represent "best guesses" in the face of a lack of more precise information. In fact, in 1906, Brown's batting average was .204.

3. height(Pee-Wee-Reese) = 6-1. This represents another default inference. Notice here that because we get to it first, the more specific fact about the height of baseball players overrides a more general fact about the height of adult males.
4. bats(Three-Finger-Brown) = Right. To get a value for the attribute *bats* required going up the *isa* hierarchy to the class *Baseball-Flayer*. But what we found there was not a value but a rule for computing a value. This rule required another value (that for *handed)* as input. So the entire process must be begun again recursively to find a value for *handed.* This time, it is necessary to go all the way up to *Person* to discover that the default value for handedness for people is *Right.* Now the rule *far bats cm* be applied, producing the result *Right.* In this case, that turns out to be wrong, since Brown is a switch hitter (i.e., he can hit both left-and right-handed).

3. Inferential Knowledge

Property inheritance is a powerful form of inference, but it is not the only useful form. Sometimes all the power of traditional **logic** (and sometimes even more than that) is necessary to describe the **inferences** that are needed. Figure 6 shows two examples of the use of **first-order predicate logic** to represent additional knowledge about baseball.

Of course, this knowledge is useless unless there is also an **inference procedure** that can exploit it (just as the default knowledge in the previous example would have been useless without our algorithm for moving through the knowledge structure). The required inference procedure now is one that **implements the standard logical rules of inference**. There are many such procedures, some of which reason forward from given facts to conclusions, others of which reason backward from desired conclusions to given facts. One of the most commonly used of these procedures *is **resolution***, which exploits a proof by **contradiction** strategy.

$$\forall x : Ball(x) \wedge Fly(x) \wedge Fair(x) \wedge Infield\text{-}Catchable(x) \wedge Occupied\text{-}Base(First) \wedge Occupied\text{-}Base(Second) \wedge (Outs < 2) \wedge \neg[Line\text{-}Drive(x) \vee Attempted\text{-}Bl\ ,\ (x)] \rightarrow Infield\text{-}Fly(x)$$

$$\forall x, y : Batter(x) \wedge batted(x, y) \wedge Infield\text{-}Fly(y) \rightarrow Out(x)$$

Figure 6: Inferential Knowledge

Recall that we hinted at the need for something besides stored primitive values with the *bats* attribute of our previous example. Logic provides a powerful structure in which to describe relationships among values. It is often useful to combine this, or some other powerful description language, with an *isa* hierarchy. In general, in fact, all of the techniques we are describing here should be viewed as building blocks of a complete representational system.

4. Procedural Knowledge

So far, our examples of baseball knowledge have concentrated on relatively static, declarative facts. But another, equally useful, kind of knowledge is operational or **procedural knowledge** that specifies what to do when. Procedural knowledge can be represented in **programs** in many ways. The most common way **is simply as code** (in some programming language such as LISP) for doing something. The machine uses the knowledge when **it executes the code to perform a task**. Unfortunately, this way of representing procedural knowledge gets low scores with respect to the properties of **inferential adequacy** (because it is very difficult to write a program that can reason about another program's behavior) and **acquisitional effici**ency (because the process of updating and debugging large pieces of code becomes unwieldy).

The most commonly used technique for representing procedural knowledge in AI programs is the use of **production rules**. Figure 7 shows an example of a production rule that represents a piece of operational knowledge typically possessed by a baseball player. Production rules, particularly ones that are augmented with information on how they are to be used, are more procedural than are the other representation methods discussed in this chapter. But making a clean distinction between **declarative** and **procedural** knowledge is difficult. The important difference is in **how the knowledge is used by the procedures that manipulate it.**

If: ninth inning, and
score is close, and
less than 2 outs, and
first base is vacant, and
batter is better hitter than next batter,
Then: walk the batter.

Figure 7: Procedural Knowledge as Rules

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.

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. What does matter is that they represent class membership and class inclusion and that class inclusion is transitive.

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 Figure 4, 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.

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 Figure 4. 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 Figure 4 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***.

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.

3. Choosing the Level 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 of ones covering a range of levels? A brief example will illustrate the problem. Suppose we are interested in the following fact:

John spotted Sue

We could represent this as 2

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) → 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),
time- span (briefly))

In this representation, we have broken the idea of *spotting* apart into more primitive concepts of *seeing* and time span. 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 A.I. programs, including those described in are based on knowledge bases described in terms of a small number of primitives. However, there are a couple of major drawbacks to this approach.

The most important **argument against** the use of low-level primitives is that **a lot of work must be done to convert each high-level fact into its primitive form**. And for many purposes, this detailed primitive representation may be unnecessary. Both in understanding language and in interpreting the world that we see, many things appear that turn out later to be irrelevant. For the sake of efficiency, it may be desirable to store these things at a very high level and then to analyze in detail only those inputs that appear to be important.

A **second problem** with the use of low-level primitives is that **simple high-level facts may require a lot of storage when broken down into primitives**. Much of that storage is really wasted since the low-level rendition of a particular high-level concept will appear many times, once for each time the high-level concept is referenced. For example, suppose that actions are being represented as combinations of a small set of primitive actions.

A **third problem** with the use of primitives is that in many domains, it is not at all clear **what the primitives should be**. And even in domains in which there may be an obvious set of primitives, there may not be enough information present in each use of the high-level constructs to enable them to be converted into their primitive components. When this is true, there is no way to avoid representing facts at a variety of levels.

The classical example of this sort of situation is provided by **kinship terminology**. There exists at least one obvious set of primitives: **mother, father, son, daughter, and possibly brother and sister**. But now suppose we are told that Mary is Sue's cousin. An attempt to describe the cousin relationship in terms of the primitives could produce any of the following interpretations:

• Mary = daughter(brother(mother(Sue)))

• Mary = daughter(sister(mother(Sue)))

• Mary = daughter(brother(father(Sue)))

• Mary = daughter(sister(father(Sue)))

Since in general we may have no way of choosing among these representations, we have no choice but to represent the fact using the non primitive relation cousin. The other way to solve this problem is to change our primitives. We could use the set: parent, child, sibling, male, and female. Then the fact that Mary is Sue's cousin could be represented as

Mary = child(sibling(parent(Sue)))

But now the primitives incorporate some generalizations that may or may not be appropriate. The main point to be learned from this example is that even in very simple domains, the correct set of primitives is not obvious.

4. Representing sets of objects

It is important to be able to represent sets of objects for several reasons. One is that there are some properties that are true of sets that are not true of the individual members of a set. As examples, consider the assertions that are being made in the sentences "There are more sheep than people in Australia" and "English speakers can be found all over the world." The only way to represent the facts described in these sentences is to attach assertions to the sets representing people, sheep, and English speakers, since, for example, no single English speaker can be found all over the world. The other reason that it is important to be able to represent sets of objects is that if a property is true of all (or even most) elements of a set, then it is more efficient to associate it once with the set rather than to associate it explicitly with every element of the set. We have already looked at ways of doing that, both in logical representations through the use of the universal quantifier and in slot-and-filler structures, where we used nodes to represent sets and inheritance to propagate set-level assertions down to individuals. As we consider ways to represent sets, we will want to consider both of these uses of set-level representations. We will also need to remember that the two uses must be kept distinct. Thus if we assert something like *large*(*Elephant*), it must be clear whether we are asserting some property of the set itself (i.e., that the set of elephants is large) or some property that holds for individual elements of the set (i.e., that anything that is an elephant is large).

Batter in our logical representation. This simple representation does make it possible to associate predicates with sets. But it does not, by itself, provide any information about the set it represents. It does not, for example, tell how to determine whether a particular object is a member of the set or not.

There are two ways to state a definition of a set and its elements. The first is to list the members. Such a specification is called an *extensional* definition. The second is to provide a rule that, when a particular object is evaluated, returns true or false depending on whether the object is in the set or not. Such a rule is called an *intensional* definition. For example, an extensional description of the set of our sun's planets on which people live is $\{Earth\}$. An intensional description is

$$\{x : sun\text{-}planet(x) \wedge human\text{-}inhabited(x)\}$$

For simple sets, it may not matter, except possibly with respect to efficiency concerns, which representation is used. But the two kinds of representations can function differently in some cases.

One way in which extensional and intensional representations differ is that they do not necessarily correspond one-to-one with each other. For example, the extensionally defined set $\{Earth\}$ has many intensional definitions in addition to the one we just gave. Others include:

$\{x : sun\text{-}planet(x) \wedge nth\text{-}farthest\text{-}from\text{-}sun(x, 3)\}$
$\{x : sun\text{-}planet(x) \wedge nth\text{-}biggest(x, 5)\}$

Thus, while it is trivial to determine whether two sets are identical if extensional descriptions are used, it may be very difficult to do so using intensional descriptions.

Intensional representations have two important properties that extensional ones lack, however. The first is that they can be used to describe infinite sets and sets not all of whose elements are explicitly known. Thus we can describe intensionally such sets as prime numbers (of which there are infinitely many) or kings of England (even though we do not know who all of them are or even how many of them there have been). The second thing we can do with intensional descriptions is to allow them to depend on parameters that can change, such as time or spatial location. If we do that, then the actual set that is represented by the description will change as a function of the value of those parameters. To see the effect of this, consider the sentence, "The president of the United States used to be a Democrat," uttered when the current president is a Republican. This sentence can mean two things. The first is that the specific person who is now president was once a Democrat. This meaning can be captured straightforwardly with an extensional representation of "the president of the United States." We just specify the individual. But there is a second meaning, namely that there was once someone who was the president and who was a Democrat. To represent the meaning of "the president of the United States" given this interpretation requires an intensional description that depends on time. Thus we might write *president*(t), where *president* is some function that maps instances of time onto instances of people, namely U.S. presidents.

5. Finding the Right Structures as Needed

For example, suppose we have a **script** (a **description** of a class of events in terms of contexts, participants, and sub events) that describes the typical sequence of events in a restaurant. 4 This script would enable us to take a text such as John went to Steak and Ate last night. He ordered a large rare steak, paid his bill, and left.

and answer *yes* to the question

Did John eat dinner last night?

Notice that nowhere in the story was John's eating anything mentioned explicitly. But the fact that when one goes to a restaurant one eats will be contained in the restaurant script. If we know in advance to use the restaurant script, then we can answer the question easily. But in order to be able to reason about a variety of things, a system must have many scripts for everything from going to work to sailing around the world. How will it select the appropriate one each time? For example, nowhere in our story was the word *restaurant* mentioned.

In fact, in order to have access to the right structure for describing a particular situation, it is necessary to solve all of the following problems:

• How to perform an **initial selection** of the most appropriate structure

• How to fill in appropriate **details** from the current situation

• How to find a **better structure** if the one chosen initially turns out not to be appropriate

• What to do if none of the **available structures** is **appropriate**

• When to **create** and **remember** a **new structure** used

This is often a very difficult question and there are several ways in which it can be answered. Three important approaches are the following:

• **Index the structures directly by the content words**. For example, let each **verb** have associated with it a structure that **describes** its **meaning**. Even for selecting simple structures, such as those representing the meanings of individual words, though, this approach may not be adequate, since many words may have several distinct meanings. For example, the word *fly* has a different meaning in each of the following sentences:

John flew to New York. (He rode in a plane from one place to another.)

John flew a kite. (He held a kite that was up in the air.)

John flew down the street. (He moved very rapidly.)

John flew into a rage. (An idiom)

• **Consider each content word in the text as a pointer to all of the structures (such as scripts) in which it might be involved.** This produces several sets of prospective structures. For example, the word *steak* might point to two scripts, one for restaurant and one for supermarket. The word *bill* might point to a restaurant and a shopping script. Take the intersection of those sets to get the structure(s), preferably precisely one, that involves all of the content words. Given the pointers just described and the story about John's trip to Steak and Ale, the restaurant script would be evoked.

One important problem with this method is that if the text contains any even slightly extraneous words, then the intersection of their associated structures will be

empty. This might occur if we had said, for example, "John rode his bicycle to Steak and Ale last night." Another problem is that it may require a great deal of computation to compute all of the possibility sets and then to intersect them. However, if computing such sets and intersecting them could be done in parallel, then the time required to produce an answer would be reasonable even if the total number of computations is large.

• **Locate one major clue in the text and use it to select an initial structure**. As other clues appear, use them to **refine** the **initial selection** or to make a completely new one if necessary. The major problem with this method is that in some situations there is not an **easily identifiable major clue**. A second problem is that it is necessary to **anticipate which clues** are going to be **important** and which are not. But the relative importance of clues can change dramatically from one situation to another. For example, in many contexts, the color of the objects involved is not important. But if we are told that "The light turned red," then the color of the light is the most important feature to consider.

None of these proposals seems to be the complete answer to the problem. It often turns out, unfortunately, that the more complex the knowledge structures are, the harder it is to tell when a particular one is appropriate.

Logic

Logic is a language for reasoning. It is a collection of rules we use when doing logical reasoning. Human reasoning has been observed over centuries from at least the times of Greeks, and patterns appearing in reasoning have been extracted, abstracted, and streamlined. The foundation of the logic we are going to learn here was laid down by a British mathematician George Boole in the middle of the 19th century, and it was further developed and used in an attempt to derive all of mathematics by Gottlob Frege, a German mathematician, towards the end of the 19th century. A British philosopher/mathematician, Bertrand Russell, found a flaw in basic assumptions in Frege's attempt but he, together with Alfred Whitehead, developed Frege's work further and repaired the damage. The logic we study today is more or less along this line.

In logic we are interested in true or false of statements, and how the truth/falsehood of a statement can be determined from other statements. However, instead of dealing with individual specific statements, we are going to use symbols to represent arbitrary statements so that the results can be used in many similar but different cases. The formalization also promotes the clarity of thought and eliminates mistakes.

There are various types of logic such as logic of sentences (propositional logic), logic of objects (predicate logic), logic involving uncertainties, logic dealing with fuzziness, temporal logic etc. Here we are going to be concerned with propositional logic and predicate logic, which are fundamental to all types of logic.

Representing Simple Facts in Logic

We can begin exploring one particular way of representing facts, the language of logic. 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 A.I. techniques were explored was mechanical theorem proving, by which was meant proving statements in various areas of mathematics, such as number theory or geometry. Mathematical theorem proving is still an active area of A.I. research. But, as we will 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.

Let's first explore the use of propositional logic as a way of representing the sort of world knowledge that an A.I. system might need. Propositional logic is appealing because it is simple to deal with and there exists a decision procedure for it. We can easily represent real-world facts as logical ***propositions*** **written as** ***well-formed formulas (wffs)*** in propositional logic.

Using these propositions, we could, for example, deduce that it is not sunny if it is raining.

“It is raining” can be represented in propositional logic as RAINING

“It is sunny” can be represented in propositional logic as SUNNY

“It is windy” can be represented in propositional logic as WINDY

“If it is raining then it is not sunny” can be represented in propositional logic as RAINING $\rightarrow$ ~SUNNY

Figure 8: Some Simple Facts in Propositional Logic

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

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. We are in even more difficulty if we try to represent the equally classic sentence

All men are mortal

because now we really need 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 **predicate logic** 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 was that if we used logical statements as a way of representing knowledge, then we had 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 they are not guaranteed to halt if the proposed statement is not a theorem. A simple such procedure is to use the rules of inference to generate theorems 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 A.I., 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 non theorem. 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 A.I. system might need.

Introduction to Propositional Logic

Propositional logic is a logic at the sentential level. The smallest unit we deal with in propositional logic is a sentence. We do not go inside individual sentences and analyze or discuss their meanings. We are going to be interested only in true or false of sentences, and major concern is whether or not the truth or falsehood of a certain sentence follows from those of a set of sentences, and if so, how. Thus sentences considered in this logic are not arbitrary sentences but are the ones that are true or false. These kinds of sentences are called **propositions**.

Statements

Statements or **propositions** are atomic components which can be more or less complex grammatical constructions expressing facts. They form the foundation of propositional logic.

Definition

An essential property of logical statements is that they have a definite truth value. Generally each statement is either **TRUE** or **FALSE**. However, as in many fields of science are there exceptions caused by incomplete or ambiguous enunciations. These facts nevertheless can count as statements, but they do not have a definite truth value.

Examples for statements

The following sentences all are statements regardless of its truth value!

- ***The moon is made of blue cheese.***
- ***Paris is the capital of France.***
- ***Mice chase cats.***
- ***He is older than she is.***

Counter examples

To understand what a statement or proposition is, it is may be a good way to show which kinds of sentences do not count as a statement.

1. A question is definitely **not** a statement
   - ***How old are you?***
   - ***Do you love me?***

In some cases perhaps it is possible to answer the question with YES or No, but it makes no sense to speak of any truth value. Both examples are not statements which can be TRUE or FALSE.

2. Commands do **not** count as a statement!

- ***Shut the window!***
- ***Please turn right!***

3. Wishes can**not** be a statement

- ***If only I hadn't said her that I'm rich.***
- ***Merry Christmas and a Happy New Year!***

Exceptions

Imagine you get a grammatical construction like the following one:

I have always told lies.

It seems to be a statement, but what is its truth value? As soon as I say it is true and it is really true, the statement changes its truth value to false, because I didn't tell a lie. If I say it is false that means if I lie, the truth value keeps true but my answer is wrong. **Conclusion:** We can say this is a statement, but we get a problem to assign a truth value to it.

Another ambiguous example is the following one:

Andy is older than Kathrin.

This appears only as a statement if somewhere in the context is explained who exactly Andy and Kathrin are. If we choose pronouns instead of the names, we get the same problem:

He is older than she is.

Such a statement has no truth value until we describe who he or she are. If we explain the coherences and put this together with the statement without a definite truth value then we get a **real statement**.

Andy is in the same class as Kathrin and he is older than she is.

This is a real statement and the people who know the circumstances are able to determine the truth value of this statement.

The Truth Functions

First of all we want to have a look at **all logic functions** considered in this chapter. In Table 1 you can see a list of all usual connectives used in logic. AND, OR and NOT are the basic functions, the remaining ones are special combinations compiled of the basic functions and simply replaced by a single truth-functional connective (*because of its importance*). Knowledge about the logic functions are necessary in many fields of science e.g. in the Design of digital circuits.

One possible **description of a logic function** is a definition with words, like "If ... then ..." or something like that. You will encounter some definitions of that kind later.

Another way of presenting this rule is by means of a **truth table**. We don't need to subscribe the function in words, we can do it the way most of the tables show. Each row of the truth table corresponds to one of the four possible combinations of truth values which may be possessed by interpretations of the schematic letters *a* and *b*. The last entry gives the truth value of the corresponding interpretations of *a* and *b*. As an example: The schema *a*∧*b* is called the *conjunction* of the two schemas *a* and *b*. Conversely, *a* and *b* are called the *conjuncts* of *a* and *b*.

Shortcut	**Symbol**	**Description**
AND	∧	*Conjunction*
OR	∨	*Disjunction*
XOR		*Exclusive Disjunction*
NOT	¬	*Negation*
NAND		*Negated Conjunction*
NOR		*Negated Disjunction*
	→	*Material Implication*
	↔	*Material Equivalence*
***Table1:** Logic functions*		

1. **Conjunction**

Let's have a look at the following sentence:

Yesterday you went to the post office and I called Anne at the phone box.

Everyday we use such expressions. The sentence expresses two main ideas. We could divide it into separate statements, the phrase *"Yesterday you went to the post office"* and *"I called Anne at the phone box"*. We join together two smaller statements by means of the link-word "**and**".

Assuming we consider statement (1) with the link-word "**and**". The sentence is true if both phrases are true. That means, if you (*apart from the fact who "you" actually is*) went yesterday to the post office **and** I called Anne at the phone box, then this sentence has the truth value "**true**". But if the answer to either or both of them is "no", then the whole sentence is **false**. Merely knowing the truth values of the two (or more) components is enough to determine the truth value of the compound statement.

"Link-words" are said to be **truth-functional** whenever the truth value of the statement obtained by using it to link two given statements is uniquely determined by the truth values of the two statements themselves.

"An interpretation satisfies the statement *a*∧*b* if and only if both *a* and *b* are satisfied."

a	**B**	**a∧b**
false	false	**false**
false	true	**false**
true	false	**false**
true	true	**true**

Table2: *The Conjunction*

Table 2 illustrates the truth table of the conjunction. As expected the compound statement is only true if both of the input variables (in our case *a* and *b*) are true.

2. **Disjunction**

The link-word "**or**" may cause slight complications in the meaning of the english language. In practice, the sense of "or" most often encountered in logic is the inclusive "or". That means: "*a* or *b* or both". To demonstrate this we will have a look at the statement: "Mike is in Paris or in Berlin". Only one of both phrases could be true, they exclude each other. Thus, this is the exclusive version of "or" which is different to the logical "or". It should just demonstrate that there is a lot of differences between the language and the logical meaning, so don't mix them up.

a	**b**	**a∨b**
false	false	**False**
false	true	**true**
true	false	**true**
true	true	**True**

Table3:*The Disjunction*

The connective for "or" is called **disjunction** and the function can be explained as follows:

"An interpretation satisfies the statement *a*∨*b* if and only if at least one of *a* and *b* is satisfied."

In other words we can say: **"An interpretation falsifies *a*∧*b* if and only if it falsifies both *a* and *b*"**. The statement "*a*∨*b*" is called the **disjunction** of *a* and *b*. The truth table corresponding to this rule is shown in Table 3.

As an example we might have a look at the phrase "Peter ate fish or Susan ate steak". Both expressions could be true at the same time. If we apply the truth values of the "or" function to the sentence, we notice that the whole statement is true in 3 cases: At first if Peter actually ate fish and Susan didn't had steak, furthermore if Peter didn't ate fish and Susan had a steak and finally if both is true, Peter really ate fish and Susan had actually steak to dinner.

3. **Negation**

The negation has much the same meaning as the English word "not". For example, **"John is not married"**. Look at the following two statements:

1. **John is married.**
2. **John is not married.**

This both statements cannot both be true, and they cannot both be false; exactly one of them must be true. The first statement is valid and only valid if the second statement is **not** valid.

	a
alse	**rue**
rue	**alse**

***Table5:** The Negation*

Note that "not" is not a link-word in the sense of joining together two statements; rather it acts on a single statement to produce a new one. A single statement with "not" in is, usually, completely determined by truth value of the statement obtained by leaving out the "not" ---- namely, it is the opposite truth value. A more strictly truth-functional expression is **"It is not the case that"**, which behaves very much as we want negation to behave in the propositional calculus. Thus instead of **"John is not married."** we could say **"It is not the case that John is married"**, in which there is a clear separation between the negation and what is negated. Sometimes, it is *necessary* to use "It is not the case that"-style to express the negation of a sentence, since simply inserting "not" has a different effect.

Example:
The negation of **The shop is open until midday.** is not **The shop is not open until midday** since both these statements could be false(e.g. if the shop is now shut, but will open at 11 a.m.). The true negation can only readily the expressed as **It is not the case that the shop is open until midday.**

The Negation is one of the tree basic logical operations (the others are the Conjunction and Disjunction), with them **all** statements are explainable.

"An interpretation satisfies the statement ¬a if the statement a is not satisfied."

4. **Exclusive Disjunction**

a	**B**	**a XOR b**
false	false	**false**
false	true	**true**
true	false	**true**
true	true	**false**

Table4: *The Exclusive Disjunction*

If we consider the compound statement "The earth is a cube or it is a sphere.", we notice the quite different meaning to the sentence considered in the last paragraph. It is straightforward that the earth can't be both a cube and a sphere. So this sentence expresses the "exclusive or". It excludes the case that both phrases in the statement could be true at the same time.

Table 4 illustrates the truth table of the XOR function. "XOR" is used as an abbreviation of the "eXclusive OR". For the **exclusive "or"** we use the abbreviation **XOR** *(for "exclusive or")* and define it as follows:

"An interpretation satisfies the statement a XOR b if and only if exactly one of a and b is satisfied."

5. **Negated Conjunction**

The negated conjunction is a combination of conjunction and negation. At first the statements *a* and *b* would be used with the conjunction. Then the result of this logical operation would be the parameter of the negation. The result of the negation is also the result of the whole negated conjunction. Sometimes would be used a special abbreviation **NAND** (consist of **N**OT and **AND**) for the negated conjunction. The ***a* NAND *b*** (NAND

is the shortcut for the negated conjunction) can be explained by **NOT(*a* AND *b*)** or **¬ (*a*⊼*b*)**.

a	**b**	**a NAND b**
false	false	**true**
false	true	**true**
true	false	**true**
true	true	**false**

Table6: *The Negated Conjunction*

6. **Negated Disjunction**

The negated conjunction is a combination of disjunction and negation. The **a NOR b** (NOR is the shortcut for the negated disjunction) can be explained by **¬(a⊻b)**.

a	**b**	**a NOR b**
false	false	**true**
false	true	**false**
true	false	**False**
true	true	**false**

Table7: *The Negated Disjunction*

7. **Material Implication**

The material implication's are typical if-statements.

Look at the following statements:

1. **If I reach the train then I arrive at 8 p.m.**
2. **I arrive at 8 p.m. if I reach the train.**
3. **When I reach the train then I arrive at 8 p.m.**

All this statements have the same meaning and contains a conclusion. If something valid them something else is also valid. Some frequently used keywords for the material implication are if, then and when. The material implication is, contrary the other logic

operations, **non** symmetric. This mean that you can **not** switch the operators of this operation without change the content of the statement.

"An interpretation satisfies the statement a →b if if the statement a is satisfied then b is also satisfied else it's always satisfied."

a	**b**	**a →b**
false	false	**true**
false	true	**true**
true	false	**false**
true	true	**true**

Table8: *The Material Implication*

8. Material Equivalence

a	**b**	**a ↔b**
false	false	**true**
false	true	**false**
true	false	**false**
true	true	**true**

Table9: *The Material Equivalence*

The material equivalence is a material implication in both directions. So you can **a ↔ b** explained by **(a →b)∧(b →a)**. So the material equivalence means that the two statements, which are connected, have the same truth status. Some examples:

1. **Tom go to the party if and only if Jane goes also to the party.**
2. **If Bill work then Mike also work and If Mike work then Bill also work.**

"An interpretation satisfies the statement a ↔b if and only if the statement's a and b has the same truth status."

Expressions and Formulas

The understanding of making formulas is important for the transformation from verbal statements into formulas. The abbreviation of formulas improves the legibility of formulas.

Propositional formulas will be formed from variables and operators by definite rules. Not every symbol combination made of this symbols is a proposional formula. Following definition explains what is a propositonal formula:

1. **Each truth value and each variable are formulas.**
2. **For every formula X , also ¬X is a formula.**
3. **For all formulas X and Y, also (X∧Y), (X∨Y), (X →Y), and (X ↔Y) are formulas.**
4. **Only those expressions are formulas which are formulas because of rule 1, 2 or 3.**

Example of a formula:

((((a∧¬b)∨c) →d) ↔¬(a∨∧c)))

This is not a formula:

((∧a) →¬b)

Priorities of the operations:

negation	**5**
conjunction	**4**
disjunction	**3**
conclusion	**2**
equivalence	**1**

Operations with higher priority have to be solved at first. To reduce the number of symbols and shape this more clearly we enter into following agreements:

1. **Parenthesis can be omitted if they can be reconstructed by the priorities of the operations.**
2. **The conjunction operation may be omitted if no conflicts arise.**

(3. For every formula X, also (X) is a formula.)

Descriptive example

complete formula:	**((((a∧b)∧c)∨((¬a∧b) ∧c))∨((a∧¬b)∧c))**
without parenthesis:	**a∧b∧c∨¬a∧b∧ ∨a∧¬b∧c**

without operator ∧: abc∨¬abc∨a¬bc

Example of a formula with abbreviations:

a(¬b)∨c →d ↔¬(a∨bc)

Creating Formulas And Terms

For the analysis of verbal statements, it can be necessary to give it the shape of a formula. The following table includes some statements which can be transformed directly into a formula.

a∧b	***a* and *b* are valid** **both *a* and *b* are valid** ***a* and *b* are valid at the same time**
a∨b	***a* or *b* are valid** ***a* and/or *b* are valid** **(at least) one of *a* and *b* is valid**
a(¬b)∨(¬a)b	**either *a* or *b* is valid** **exactly one of *a* and *b* is valid** ***a* exclusive-or *b*** ***a* and *b* ar not equivalent**
a →b	***b* is valid, if *a* is valid** **if *a* is valid, also *b* is valid** **if *a* is valid, so *b* is valid also** **from a follows b** ***a* is sufficient for *b*** ***b* is necessary for *a***
a ↔b	***a* is equivalent to *b*** ***a* is exactly valid if *b* is valid** ***a* is only valid if *b* is valid** ***a* is sufficient and necessary for *b*** ***a* is valid if and only if *b* is valid**
¬a	***a* is not valid** **the negation of *a* is valid**

The creation of formulas is possible by following method.

Example 1: If the thief flees, the dog barks, else not.

1. **Discover the statements and assign variables to them.**
   t...the thief flees
   d...the dog barks
2. **If necessary, transform the verbal form into an equivalent form from the table above.**

Only if **the thief flees,** ***then*** **the dog barks.**

3. Find out the underlying formula.
a **is valid only if** ***b*** **is valid**
a ↔b

4. Combine the formula of number 3 with the variables of number 1.
t ↔d

Example 2:

If ***it is sunday*** **and** ***nice weather, we go swimming*****.**

s...it is sunday
w...it is nice weather
g...we go swimming

s^w →g

Assignment of Variables

Without knowing which values can be assigned to a variable, it is difficult to say, if a statement is correct or not. Variables are symbols, which stand for any object of a definite sphere in terms, expressions and formulas. The symbols of variables are letters (a, b, c ...). An assignment is a function which allocates a truth value to each variable. Sometimes the assignment is called interpretation, too. Instead of the values "true" and "false" often the digits "1" respectively "0" are used.

Value of a formula

The value of a formula can be calculated by replacing the variables by their values and using the logical operators (negation, conjunction, disjunction, implication, and equivalence).

General way:

1. Replace the variables by their values given by the assignment
2. Calculate the Value of the formula using the logical operator

Example:

- Formula: (a → b) c ↔ ab ⩒ d
- Values of the variables: a=F b=T c=F d=T

1. Insert the Values in the formula:
   - (F → T) F ↔F ∧ T ⩒ T
2. Calculate the Value: (step by step)
   - T∧ F↔F ⩒ T
   - F ↔ T

- F

a	**b**	**c**	**d**	**a(¬b)∨c →d**	**¬(a∨bc)**
F	**F**	**F**	**F**	T	T
F	**F**	**F**	**T**	T	T
T	**F**	**T**	**F**	F	F
T	**T**	**T**	**T**	T	F

Satisfiability, Validity and Equivalence

Satisfiable formulas:

If there exist at least one interpretation for which it is true then it will be called a satisfiable formula. The value of the formula is true for at least one assingment. It plays no rule how many models the formula has.

Example: Satisfiable formulas

assignment	**a**	**b**	**a →b**	**a∧b**	**¬(a ↔b)**	**¬(a∨b)**	**a∨(¬a)**
A	F	F	T	F	F	T	T
B	F	T	T	F	T	F	T
C	T	F	F	F	T	F	T
D	T	T	T	T	F	F	T

All these formulas are called satisfiable, because they have at least one model.

Valid formulas (tautologies):

*A formula is called **valid** (or **tautology**) if all assignments are models of this formula.* The value of the formula is true for all assignments. If a tautology is part of a more complex formula then you could replace it by the value 1.

Some easy tautologies:

- **1∨1**
- **a∨(¬a)**
- **0∨a ↔a**
- **1∧a ↔a**

All formulas are true for every possible assignment.

Famous and useful tautologies:

- **a(a →b) →b**

- **(a →b)(¬b) →a**
- **b((¬a) →(¬b)) →(¬a)**
- **(a →b)(b →c) →(a →c)**
- **(a →b) ↔((¬b) →(¬a))**
- **a →(b →a)**
- **(¬a) →(a →b)**
- **(a →b)((¬a) →b) →b**
- **(a →b)(a →(¬b)) →(¬a)**
- **ab∨c →b∨c**

Unsatisfiable formulas (contradictories):

*A formula is called **unsatisfiable** (or **contradictory**), if there does not exist a model of it.* he value of the formula is false for all assignments. If a contradictory is part of a more complex formula then you could replace it by the value 0. The negation of a valid formula is a unsatisfiable one: If X is a tautology then is (¬X) a contadictory.

Some easy contradictories:

- **0∧1**
- **a∧0**
- **a(¬a)**
- **((¬a)b∨a(¬b)) ↔(ab∨(¬a)(¬b))**

Equivalence of formulas:

*Two formulas X and Y are called **equivalent** (notation X=Y) if and only if for every assignment A holds: The value of formula X is equal to the value of formula Y for the assignment **A (Value(X,A)=Value(Y,A)).*** That means for every formulas X and Y:If and only if X=Y holds then X ↔Y is a valid formula.
The easy way to proof the equivalence of two formulas is to write them in the same table and compare the values of these formulas for every row (assignment).

Some easy equivalences:

- **0∧1 = 0∧0**
- **a∨b = ¬(¬b)∨a**

Famous and useful equivalences

¬0 = 1		**¬1 = 0**	
0∧a = 0	**1∧a = a**	**0∨a = a**	**1∨a = 1**
(¬a)a = 0		**(¬a)∨a = 1**	

$a \rightarrow b = (\neg a) \vee b$	**$a \leftrightarrow b = (a \rightarrow b)(b \rightarrow a)$**		
Idempotency:	**$aa = a$**	**$a \vee a = a$**	
Double Negation:	**$\neg(\neg a) = a$**		
Commutativity:	**$ab = ba$**	**$a \vee b = b \vee a$**	**$a \leftrightarrow b = b \leftrightarrow a$**
Associativity:	**$(ab)c = a(bc)$**	**$(a \vee b) \vee c = a \vee (b \vee c)$**	
Distributivity:	**$(a \vee b)c = ac \vee ab$**	**$(ab) \vee c = (a \vee c)(b \vee c)$**	
Laws of deMorgan:	**$\neg(a \vee b) = (\neg a)(\neg b)$**	**$\neg(ab) = (\neg a) \vee (\neg b)$**	
Absorption:	**$a(a \vee b) = a$**	**$a \vee ac = a$**	

There are also propositions that are always false such as ($P \wedge \neg P$). Such a proposition is called a **contradiction**. A proposition that is neither a tautology nor a contradiction is called a **contingency**. For example ($P \vee Q$) is a contingency. For the proposition $P \rightarrow Q$, the proposition $Q \rightarrow P$ is called its **converse**, and the proposition $\neg Q \rightarrow \neg P$ is called its **contrapositive**.

For example for the proposition "If it rains, then I get wet",

Converse: If I get wet, then it rains.

Contrapositive: If I don't get wet, then it does not rain.

The converse of a proposition is not necessarily logically equivalent to it, that is they may or may not take the same truth value at the same time. On the other hand, *the contrapositive of a proposition is always logically equivalent to the proposition*. That is, they take the same truth value regardless of the values of their constituent variables. Therefore, "If it rains, then I get wet." and "If I don't get wet, then it does not rain." are logically equivalent. If one is true then the other is also true, and vice versa.

Two statements are ***consistent*** if and only if their conjunction is not a contradiction. Obviously truth tables are adequate to test validity, tautology, contradiction, contingency, consistency, and equivalence. This is important because truth tables require no ingenuity or insight, just patience and the mechanical application of rules. Truth tables correctly constructed will always give us the right answer.

There are two methods for proofing the validity of a formula:

Version 1 (semantic method):

Calculate the truth value of the formula for each assignment.

Version 2 (syntactic method):

Derive the formula from formulas known as valid in a formal (syntactic) manner.

Propositional calculus

In semantic method we will be creating tables and analyzing the value of the formula for each assignment. This is a very easy method for formulas with 2 or 3 different variables. But let's imagine a formula with 6 variables. If you will use a value table then you have to calculate the value for 64 (!) different assignments. That will take a lot of time. Even this problem solves the propositional calculus. You don't look into the value for a special interpretation. Using rules you derive the formula to a easier one, instead. In the propositional calculus there are two rules only.

Definition

If there are some valid formulas and some formal (syntactic) rules to derive new formulas which are also valid, then this collection of basic formulas and rules is called a calculus.

Rules

Insertion rule	**Example**
Let X be a valid formula, v be a variable occuring in X, Y be any formula.	**The formula (c∨b) ↔(b∨c) is valid. The formula contains the variables b and c. (u∨v)w is a formula.**
If the variable v at all occurences simultaneously is replaced by the formula Y, then a new formula has been created. This new formula is also valid.	**Inserting the formula (u∨v)w instead of c gives the valid formula ((u∨v)w ∨b) ↔(b∨ (u∨v)w).**

Substitution rule	**Example**
Let X be any formula, Y be a formula occuring in the formula X (as subformula), Y ↔Z be a valid formula.	**¬(a∨b)ac∨(a∨b)c is a formula containing the formula ¬(a∨b). ¬(a∨b) is equivalent to (¬a)(¬b),**

	formula ¬(a∨b) ↔(¬ a)(¬b) is valid.
If the formula Y is replaced by Z (at least) at one occurence in the formula X then a new formula is created. This new formula is equivalent to the original formula X. That means, if W is the new formula then W ↔X is valid.	**Because of the desciped conditions the formula (¬a)(¬b)ac ∨(a∨b)c ↔ ¬(a∨b)ac∨ (a∨b)c is valid. That means: (¬a)(¬b)ac ∨(a∨b)c is equivalent to ¬(a∨b)ac ∨(a∨b)c.**

Soundness and completeness of the propositional calculus:

Assuming the above mentioned set of famous and useful equivalences as valid formulas (the symbols = replaced by the operations ↔) and using the insertion and the substitution rule only, then only valid formulas can be derived (soundness). Additionally, each valid formula can be derived (completeness).

Axiom System

The basic **set of valid formulas** is called **axiom system.** The above mentioned set of famous and useful equivalences represents only one but practicable axiom system which is complete and sound.

Examples:

1) Creating a valid formula

Step	Transformation	Result
a)	**The initial formula is the term 1**	**1**
b)	**Using the substitution rule (SR), we replace 1 by a∨(¬a).**	**a∨(¬a)**
c)	**With the help of the insertion rule (IR), we change a to u∨v at all positions.**	**u∨v∨ (¬(u∨v))**
d)	**The formula ¬(u∨v) is equivalent to (¬ u)(¬v).(SR)**	**u∨v∨(¬u) (¬v)**

e)	**Now we insert ¬(xy) instead of u and x →z instead of v. (IR)**	**¬(xy)∨(x →z) ∨xy(¬(x →z))**
f)	**Because the formula (x →z) ↔(¬x)∨z is valid we get the following formula. (SR)**	**¬(xy)∨(x →z) ∨xy(¬(¬ (x)∨z))**

2) Checking the validity of a formula by deriving new and easier formulas

Step	Transformation	Result
a)	**The initial formula.**	**a →(b →a)**
b)	**x →y is equivalent to (¬x)∨y. (SR)**	**(¬a)∨((¬b)∨a)**
c)	**The associativity (SR) brings us:**	**(¬a)∨(¬b)∨a**
d)	**The formula ((¬x)∨x) ↔1 is a tautology. (SR)**	**1∨(¬b)**
e)	**1∨x is equivalent to 1 (SR)**	**1**
f)	**Interpretation**	**The formula a →(b →a) is a tautology.**

3) Proofing the equivalence of two formulas X and Y

There are two ways to proof the equivalence of formulas. The first one is to show the correctness of the statement: The formula X →Y is a tautology. I have shown this in the previous example. The second possibility is to transform one formula into the other with the help of the two rules.

The two formulas are:

X = p →q
Y = (¬q) →(¬p)

Step	Transformation	Result
a)	**The initial formula.**	**p →q**
b)	**(a →b) ↔(¬a)∨b (SR)**	**(¬p)∨q**
c)	**Commutativity (SR)**	**q∨(¬p)**

d)	**a ⇔¬(¬a) (SR)**	**¬(¬q)∨(¬p)**
f)	**Interpretation**	**The formulas X and Y are equivalent.**

Reasoning with Propositions

Logical reasoning is the process of drawing conclusions from premises using rules of inference. However, ***inference rules of propositional logic are also applicable to predicate logic*** and reasoning with propositions is fundamental to reasoning with predicate logic. These inference rules are results of observations of human reasoning over centuries. Though there is nothing absolute about them, they have contributed significantly in the scientific and engineering progress the mankind have made. Today they are universally accepted as the rules of logical reasoning and they should be followed in our reasoning.

Since inference rules are based on identities and implications, we are going to study them first. We start with three types of proposition which are used to define the meaning of "identity" and "implication". In general, a **rule of inference** is just an instruction for obtaining additional true statements from a list of true statements.

Modus Ponens

In logic, **modus ponens** (Latin: *mode that affirms*; often abbreviated **MP**) is a valid, simple argument form. It is a very common form of reasoning, and takes the following form:

If P, then Q.
P.
Therefore, Q.

In logical operator notation:

$$((P \to Q) \land P) \vdash Q$$

where ⊢represents the logical assertion (that Q is true).

The modus ponens rule may also be written:

$$\frac{(P \to Q), P}{Q}$$

The argument form has two premises. The first premise is the "if–then" or ***conditional*** claim, namely that P implies Q. The second premise is that P, the ***antecedent*** of the conditional claim, is true. From these two premises it can be logically concluded that Q, the ***consequent*** of the conditional claim, must be true as well.

Here is an example of an argument that fits the form *modus ponens*:

If today is Tuesday, then I will go to work.
Today is Tuesday.
Therefore, I will go to work.

The fact that the argument is **valid** cannot assure us that any of the statements in the argument are true; the validity of modus ponens tells us that the conclusion must be true if all the premises are true. It is wise to recall that a valid argument within which one or more of the premises are not true is called an ***unsound*** argument, whereas if all the premises are true, then the argument is ***sound***. In most logical systems, modus ponens is considered to be valid. However, the instances of its use may be either sound or unsound:

In this form, Modus Ponens is our first **rule of inference.** We shall use rules of inference to assemble lists of true statements, called **proofs.** A **proof** is a way of showing how a conclusion follows from a collection of premises. Modus Ponens, in particular, allows us to say that, if $A \rightarrow B$ and A both appear as statements in a proof, then we are justified in adding B as another statement in the proof.

Our convention has been that small letters like p stand for atomic statements. But, there is no reason to restrict Modus Ponens to such statements. For example, we would like to be able to make the following argument:

If roses are red and violets are blue, then sugar is sweet and so are you.
Roses are red and violets are blue.

Therefore, sugar is sweet and so are you.

In symbols, this is

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

$\therefore r \wedge s$

Modus ponens can also be referred to as **Affirming the Antecedent** or **The Law of Detachment**. An expanded form of the argument, called **multiple modus ponens** (often abbreviated **mmp**), also exists, and has the following form:

If P, then Q.
If Q, then R.
P.
Therefore, R.

In logical operator notation:

$$P \rightarrow Q$$
$$Q \rightarrow R$$
$$P$$
$$\vdash R$$

If the left(right) hand side of an identity appearing in a proposition is replaced by the right(left) hand side of the identity, then the resulting proposition is logically equivalent to the original proposition. Thus the new proposition is deduced from the original proposition. For example in the proposition $P \wedge (Q \rightarrow R)$, $(Q \rightarrow R)$ can be replaced with $(\neg Q \vee R)$ to conclude $P \wedge (\neg Q \vee R)$, since $(Q \rightarrow R) \Leftrightarrow (\neg Q \vee R)$

Similarly if the left(right) hand side of an implication appearing in a proposition is replaced by the right(left) hand side of the implication, then the resulting proposition is logically implied by the original proposition. Thus the new proposition is deduced from the original proposition. The tautologies listed as "implications" can also be considered **inference rules** as shown below.

Rules of Inference	Tautological Form	Name
P --- $P \vee Q$	$P \Rightarrow (P \vee Q)$	addition
$P \wedge Q$ --- P	$(P \wedge Q) \Rightarrow P$	simplification
P $P \rightarrow Q$ --- Q	$[P \wedge (P \rightarrow Q)] \Rightarrow Q$	modus ponens
$\neg Q$ $P \rightarrow Q$ --- $\neg P$	$[\neg Q \wedge (P \rightarrow Q)] \Rightarrow \neg P$	modus tollens
$P \vee Q$ $\neg P$ --- Q	$[(P \vee Q) \wedge \neg P] \Rightarrow Q$	disjunctive syllogism
$P \rightarrow Q$ $Q \rightarrow R$ --- $P \rightarrow R$	$[(P \rightarrow Q) \wedge (Q \rightarrow R)] \Rightarrow [P \rightarrow R]$	hypothetical syllogism

P Q -------- P ∧Q		conjunction
(P →Q) ∧(R →S) P ∨R ---------- Q ∨S	[(P →Q) ∧(R →S) ∧(P ∨R)] ⇒[Q ∨S]	constructive dilemma
(P →Q) ∧(R →S) ¬Q ∨¬S ---------- ¬P ∨¬R	[(P →Q) ∧(R →S) ∧(¬Q ∨¬S)] ⇒[¬P ∨¬R]	destructive dilemma

Predicate Logic

The propositional logic is not powerful enough to represent all types of assertions that are used in computer science and mathematics, or to express certain types of relationship between propositions such as equivalence. For example, the assertion "x is greater than 1", where x is a variable, is not a proposition because you can not tell whether it is true or false unless you know the value of x. Thus the propositional logic can not deal with such sentences. However, such assertions appear quite often in mathematics and we want to do inferencing on those assertions.

Also the pattern involved in the following logical equivalences can not be captured by the propositional logic:
"Not all birds fly" is equivalent to "Some birds don't fly".
"Not all integers are even" is equivalent to "Some integers are not even".
"Not all cars are expensive" is equivalent to "Some cars are not expensive",
... .

Each of those propositions is treated independently of the others in propositional logic. For example, if P represents "Not all birds fly" and Q represents "Some integers are not even", then there is no mechanism in propositional logic to find out that P is equivalent to Q. Hence to be used in inferencing, each of these equivalences must be listed individually rather than dealing with a general formula that covers all these equivalences collectively and instantiating it as they become necessary, if only propositional logic is used.

Thus we need more powerful logic to deal with these and other problems. The predicate logic is one of such logic and it addresses these issues among others.

A **predicate** is a verb phrase template that describes a property of objects, or a relationship among objects represented by the variables. For example, the sentences "The car Tom is driving is blue", "The sky is blue", and "The cover of this book is blue" come from the template "is blue" by placing an appropriate noun/noun phrase in front of

it. The phrase **"is blue"** is a predicate and it describes the property of being blue. Predicates are often given a **name**. For example any of "is_blue", "Blue" or "B" can be used to represent the predicate "is blue" among others. If we adopt B as the name for the predicate "is_blue", sentences that assert an object is blue can be represented as "B(x)", where x represents an arbitrary object. B(x) reads as "x is blue".

Similarly the sentences "John gives the book to Mary", "Jim gives a loaf of bread to Tom", and "Jane gives a lecture to Mary" are obtained by substituting an appropriate object for variables *x*, *y*, and *z* in the sentence "*x* gives *y* to *z*". The template "... gives ... to ..." is a predicate and it describes a relationship among three objects. This predicate can be represented by Give(x, y, z) or G(x, y, z), for example.

The sentence "John gives the book to Mary" can also be represented by another predicate such as "gives a book to". Thus if we use B(x, y) to denote this predicate, "John gives the book to Mary" becomes B(John, Mary). In that case, the other sentences, "Jim gives a loaf of bread to Tom", and "Jane gives a lecture to Mary", must be expressed with other predicates. Not all strings can represent propositions of the predicate logic. Those which produce a proposition when their symbols are interpreted must follow the rules given below, and they are called **wffs**(well-formed formulas) of the first order predicate logic.

We begin this section by specifying more precisely the way in which the possible worlds of first-order logic reflect the ontological commitment to objects and relations. Then we introduce the various elements of the language, explaining their semantics as we go along.

Models for first-order logic

Recall from Chapter 7 that the models of a logical language are the formal structures that constitute the possible worlds under consideration. Models for propositional logic are just sets of truth values for the proposition symbols. Models for first-order logic are more interesting. First, they have objects in them! The **domain** of a model is the set of objects it contains: these objects are sometimes called **domain elements**. Figure 8.2 shows a model with five objects: Richard the Lionheart, King of England from 1189 to 1199; his younger brother, the evil King John, who ruled from 1199 to 1215; the left legs of Richard and John; and a crown.

The objects in the model may be related in various ways. In the figure, Richard and John are brothers. Formally speaking, a relation is just the set of **tuples** of objects that are related. (A tuple is a collection of objects arranged in a fixed order and is written with angle brackets surrounding the objects.) Thus, the brotherhood relation in this model is the set

$$\{\langle \text{Richard the Lionheart, King John}\rangle, \langle \text{King John, Richard the Lionheart}\rangle\}. \quad (8.1)$$

(Here we have named the objects in English, but you may, if you wish, mentally substitute the pictures for the names.) The crown is on King John's head, so the "on head" relation contains

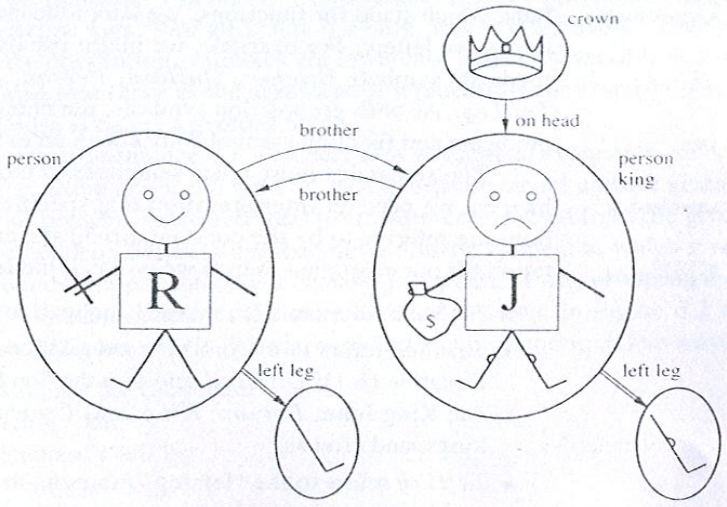


Figure 8.2 A model containing five objects, two binary relations, three unary relations (indicated by labels on the objects), and one unary function, left-leg.

just one tuple, ⟨the crown. King John⟩. The "brother" and "on head" relations are binary relations—that is, they relate pairs of objects. The model also contains unary relations, or properties: the "person" property is true of both Richard and John; the "king" property is true only of John (presumably because Richard is dead at this point); and the "crown" property is true only of the crown.

Certain kinds of relationships are best considered as functions, in that a given object must be related to exactly one object in this way. For example, each person has one left leg, so the model has a unary "left leg" function that includes the following mappings:

$$\begin{aligned} &\langle \text{Richard the Lionheart} \rangle \rightarrow \text{Richard's left leg} \\ &\langle \text{King John} \rangle \rightarrow \text{John's left leg} . \end{aligned} \tag{8.2}$$

Strictly speaking, models in first-order logic require **total functions**, that is, there must be a value for every input tuple. Thus, the crown must have a left leg and so must each of the left legs. There is a technical solution to this awkward problem involving an additional "invisible" object that is the left leg of everything that has no left leg, including itself. Fortunately, as long as one makes no assertions about the left legs of things that have no left legs, these technicalities are of no import.

Symbols and interpretations

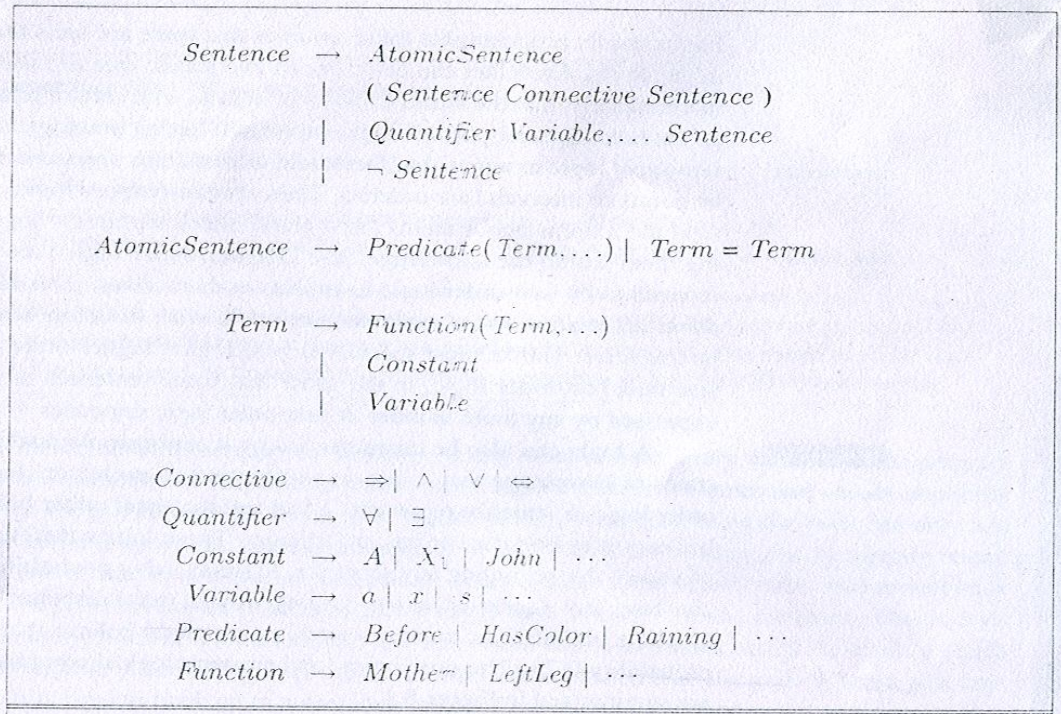
We turn now to the syntax of the language. The impatient reader can obtain a complete description from the formal grammar of first-order logic in Figure 8.3.

The basic syntactic elements of first-order logic are the symbols that stand for objects, relations, and functions. The symbols, therefore, come in three kinds: **constant symbols**, which stand for objects; **predicate symbols**, which stand for relations; and **function symbols**, which stand for functions. We adopt the convention that these symbols will begin with uppercase letters. For example, we might use the constant symbols *Richard* and *John*; the predicate symbols *Brother*, *OnHead*, *Person*, *King*, and *Crown*; and the function symbol *LeftLeg*. As with proposition symbols, the choice of names is entirely up to the user. Each predicate and function symbol comes with an **arity** that fixes the number of arguments.

The semantics must relate sentences to models in order to determine truth. For this to happen, we need an **interpretation** that specifies exactly which objects, relations and functions are referred to by the constant, predicate, and function symbols. One possible interpretation for our example—which we will call the **intended interpretation**—is as follows:

- *Richard* refers to Richard the Lionheart and *John* refers to the evil King John.
- *Brother* refers to the brotherhood relation, that is, the set of tuples of objects given in Equation (8.1); *OnHead* refers to the "on head" relation that holds between the crown and King John; *Person*, *King*, and *Crown* refer to the sets of objects that are persons, kings, and crowns.
- *LeftLeg* refers to the "left leg" function, that is, the mapping given in Equation (8.2).

There are many other possible interpretations relating these symbols to this particular model. For example, one interpretation maps *Richard* to the crown and *John* to King John's left leg. There are five objects in the model, so there are 25 possible interpretations just for the



$$
\begin{aligned}
\mathit{Sentence} &\rightarrow \mathit{AtomicSentence} \\
&\mid (\ \mathit{Sentence}\ \mathit{Connective}\ \mathit{Sentence}\ ) \\
&\mid \mathit{Quantifier}\ \mathit{Variable}, \ldots\ \mathit{Sentence} \\
&\mid \neg\ \mathit{Sentence} \\
\\
\mathit{AtomicSentence} &\rightarrow \mathit{Predicate}(\mathit{Term}, \ldots) \mid \mathit{Term} = \mathit{Term} \\
\\
\mathit{Term} &\rightarrow \mathit{Function}(\mathit{Term}, \ldots) \\
&\mid \mathit{Constant} \\
&\mid \mathit{Variable} \\
\\
\mathit{Connective} &\rightarrow \Rightarrow \mid \wedge \mid \vee \mid \Leftrightarrow \\
\mathit{Quantifier} &\rightarrow \forall \mid \exists \\
\mathit{Constant} &\rightarrow A \mid X_1 \mid \mathit{John} \mid \cdots \\
\mathit{Variable} &\rightarrow a \mid x \mid s \mid \cdots \\
\mathit{Predicate} &\rightarrow \mathit{Before} \mid \mathit{HasColor} \mid \mathit{Raining} \mid \cdots \\
\mathit{Function} &\rightarrow \mathit{Mother} \mid \mathit{LeftLeg} \mid \cdots
\end{aligned}
$$

Figure 8.3 The syntax of first-order logic with equality, specified in Backus–Naur form. (See page 984 if you are not familiar with this notation.) The syntax is strict about parentheses; the comments about parentheses and operator precedence on page 205 apply equally to first-order logic.

constant symbols *Richard* and *John*. Notice that not all the objects need have a name—for example, the intended interpretation does not name the crown or the legs. It is also possible for an object to have several names; there is an interpretation under which both *Richard* and *John* refer to the crown. If you find this possibility confusing, remember that, in propositional logic, it is perfectly possible to have a model in which *Cloudy* and *Sunny* are both true; it is the job of the knowledge base to rule out models that are inconsistent with our knowledge.

The truth of any sentence is determined by a model and an interpretation for the sentence's symbols. Therefore, entailment, validity, and so on are defined in terms of *all possible models* and *all possible interpretations*. It is important to note that the number of domain elements in each model may be unbounded—for example, the domain elements may be integers or real numbers. Hence, the number of possible models is unbounded, as is the number of interpretations. Checking entailment by the enumeration of all possible models, which works for propositional logic, is not an option for first-order logic. Even if the number of objects is restricted, the number of combinations can be very large. With the symbols in our example, there roughly 10^{25} combinations for a domain with five objects. (See Exercise 8.5.)

Terms

A **term** is a logical expression that refers to an object. Constant symbols are therefore terms, but it is not always convenient to have a distinct symbol to name every object. For example, in English we might use the expression "King John's left leg" rather than giving a name to his leg. This is what function symbols are for: instead of using a constant symbol, we use $LeftLeg(John)$. In the general case, a complex term is formed by a function symbol followed by a parenthesized list of terms as arguments to the function symbol. It is important to remember that a complex term is just a complicated kind of name. It is not a "subroutine call" that "returns a value." There is no $LeftLeg$ subroutine that takes a person as input and returns a leg. We can reason about left legs (e.g., stating the general rule that everyone has one and then deducing that John must have one) without ever providing a definition of $LeftLeg$. This is something that cannot be done with subroutines in programming languages.[4]

The formal semantics of terms is straightforward. Consider a term $f(t_1, \ldots, t_n)$. The function symbol f refers to some function in the model (call it F); the argument terms refer to objects in the domain (call them $d_1, \ldots, d_n$); and the term as a whole refers to the object that is the value of the function F applied to $d_1, \ldots, d_n$. For example, suppose the $LeftLeg$ function symbol refers to the function shown in Equation (8.2) and $John$ refers to King John, then $LeftLeg(John)$ refers to King John's left leg. In this way, the interpretation fixes the referent of every term.

Atomic sentences

Now that we have both terms for referring to objects and predicate symbols for referring to relations, we can put them together to make **atomic sentences** that state facts. An atomic sentence is formed from a predicate symbol followed by a parenthesized list of terms:

$$Brother(Richard, John).$$

This states, under the intended interpretation given earlier, that Richard the Lionheart is the brother of King John.[5] Atomic sentences can have complex terms as arguments. Thus,

$$Married(Father(Richard), Mother(John))$$

states that Richard the Lionheart's father is married to King John's mother (again, under a suitable interpretation).

An atomic sentence is ***true*** *in a given model, under a given interpretation, if the relation referred to by the predicate symbol holds among the objects referred to by the arguments.*

---

[4] **λ-expressions** provide a useful notation in which new function symbols are constructed "on the fly." For example, the function that squares its argument can be written as $(\lambda x \; x \times x)$ and can be applied to arguments just like any other function symbol. A λ-expression can also be defined and used as a predicate symbol. (See Chapter 22.) The `lambda` operator in Lisp plays exactly the same role. Notice that the use of λ in this way does *not* increase the formal expressive power of first-order logic, because any sentence that includes a λ-expression can be rewritten by "plugging in" its arguments to yield an equivalent sentence.

[5] We will usually follow the argument ordering convention that $P(x, y)$ is interpreted as "x is a P of y."

Complex sentences

We can use **logical connectives** to construct more complex sentences, just as in propositional calculus. The semantics of sentences formed with logical connectives is identical to that in the propositional case. Here are four sentences that are true in the model of Figure 8.2 under our intended interpretation:

$$\neg Brother(LeftLeg(Richard), John)$$
$$Brother(Richard, John) \wedge Brother(John, Richard)$$
$$King(Richard) \vee King(John)$$
$$\neg King(Richard) \Rightarrow King(John) .$$

Quantifiers

Once we have a logic that allows objects, it is only natural to want to express properties of entire collections of objects, instead of enumerating the objects by name. **Quantifiers** let us do this. First-order logic contains two standard quantifiers, called *universal* and *existential*.

Universal quantification ($\forall$)

Recall the difficulty we had in Chapter 7 with the expression of general rules in propositional logic. Rules such as "Squares neighboring the wumpus are smelly" and "All kings are persons" are the bread and butter of first-order logic. We will deal with the first of these in Section 8.3. The second rule, "All kings are persons," is written in first-order logic as

$$\forall x \; King(x) \Rightarrow Person(x) .$$

$\forall$ is usually pronounced "For all . . .". (Remember that the upside-down A stands for "all.") Thus, the sentence says, "For all x, if x is a king, then x is a person." The symbol x is called a **variable**. By convention, variables are lowercase letters. A variable is a term all by itself, and as such can also serve as the argument of a function—for example, $LeftLeg(x)$. A term with no variables is called a **ground term**.

Intuitively, the sentence $\forall x \; P$, where P is any logical expression, says that P is true for every object x. More precisely, $\forall x \; P$ is true in a given model under a given interpretation if P is true in all possible **extended interpretations** constructed from the given interpretation, where each extended interpretation specifies a domain element to which x refers.

This sounds complicated, but it is really just a careful way of stating the intuitive meaning of universal quantification. Consider the model shown in Figure 8.2 and the intended interpretation that goes with it. We can extend the interpretation in five ways:

- $x \rightarrow$ Richard the Lionheart,
- $x \rightarrow$ King John,
- $x \rightarrow$ Richard's left leg,
- $x \rightarrow$ John's left leg,
- $x \rightarrow$ the crown.

The universally quantified sentence $\forall x \;\; King(x) \Rightarrow Person(x)$ is true under the original interpretation if the sentence $King(x) \Rightarrow Person(x)$ is true in each of the five extended inter-

pretations. That is, the universally quantified sentence is equivalent to asserting the following five sentences:

Richard the Lionheart is a king $\Rightarrow$ Richard the Lionheart is a person.
King John is a king $\Rightarrow$ King John is a person.
Richard's left leg is a king $\Rightarrow$ Richard's left leg is a person.
John's left leg is a king $\Rightarrow$ John's left leg is a person.
The crown is a king $\Rightarrow$ the crown is a person.

Let us look carefully at this set of assertions. Since, in our model, King John is the only king, the second sentence asserts that he is a person, as we would hope. But what about the other four sentences, which appear to make claims about legs and crowns? Is that part of the meaning of "All kings are persons"? In fact, the other four assertions are true in the model, but make no claim whatsoever about the personhood qualifications of legs, crowns, or indeed Richard. This is because none of these objects is a king. Looking at the truth table for $\Rightarrow$ (Figure 7.8), we see that the implication is true whenever its premise is false—*regardless* of the truth of the conclusion. Thus, by asserting the universally quantified sentence, which is equivalent to asserting a whole list of individual implications, we end up asserting the conclusion of the rule just for those objects for whom the premise is true and saying nothing at all about those individuals for whom the premise is false. Thus, the truth-table entries for $\Rightarrow$ turn out to be perfect for writing general rules with universal quantifiers.

A common mistake, made frequently even by diligent readers who have read this paragraph several times, is to use conjunction instead of implication. The sentence

$$\forall x \;\; King(x) \wedge Person(x)$$

would be equivalent to asserting

Richard the Lionheart is a king $\wedge$ Richard the Lionheart is a person,
King John is a king $\wedge$ King John is a person,
Richard's left leg is a king $\wedge$ Richard's left leg is a person,

and so on. Obviously, this does not capture what we want.

Existential quantification ($\exists$)

Universal quantification makes statements about every object. Similarly, we can make a statement about *some* object in the universe without naming it, by using an existential quantifier. To say, for example, that King John has a crown on his head, we write

$$\exists x \;\; Crown(x) \wedge OnHead(x, John) \; .$$

$\exists x$ is pronounced "There exists an x such that . . ." or "For some x . . .".

Intuitively, the sentence $\exists x \;\; P$ says that P is true for at least one object x. More precisely, $\exists x \; P$ is true in a given model under a given interpretation if P is true in *at least one* extended interpretation that assigns x to a domain element. For our example, this means

that at least one of the following must be true:

Richard the Lionheart is a crown $\wedge$ Richard the Lionheart is on John's head;
King John is a crown $\wedge$ King John is on John's head;
Richard's left leg is a crown $\wedge$ Richard's left leg is on John's head;
John's left leg is a crown $\wedge$ John's left leg is on John's head;
The crown is a crown $\wedge$ the crown is on John's head.

The fifth assertion is true in the model, so the original existentially quantified sentence is true in the model. Notice that, by our definition, the sentence would also be true in a model in which King John was wearing two crowns. This is entirely consistent with the original sentence "King John has a crown on his head." [6]

Just as $\Rightarrow$ appears to be the natural connective to use with $\forall$, $\wedge$ is the natural connective to use with $\exists$. Using $\wedge$ as the main connective with $\forall$ led to an overly strong statement in the example in the previous section; using $\Rightarrow$ with $\exists$ usually leads to a very weak statement, indeed. Consider the following sentence:

$$\exists x \;\; Crown(x) \Rightarrow OnHead(x, John) \; .$$

On the surface, this might look like a reasonable rendition of our sentence. Applying the semantics, we see that the sentence says that at least one of the following assertions is true:

Richard the Lionheart is a crown $\Rightarrow$ Richard the Lionheart is on John's head;
King John is a crown $\Rightarrow$ King John is on John's head;
Richard's left leg is a crown $\Rightarrow$ Richard's left leg is on John's head;

and so on. Now an implication is true if both premise and conclusion are true, *or if its premise is false*. So if Richard the Lionheart is not a crown, then the first assertion is true and the existential is satisfied. So, an existentially quantified implication sentence is true in any model containing an object for which the premise of the implication is false; hence such sentences really do not say much at all.

Nested quantifiers

We will often want to express more complex sentences using multiple quantifiers. The simplest case is where the quantifiers are of the same type. For example, "Brothers are siblings" can be written as

$$\forall x \;\; \forall y \;\; Brother(x, y) \Rightarrow Sibling(x, y) \; .$$

Consecutive quantifiers of the same type can be written as one quantifier with several variables. For example, to say that siblinghood is a symmetric relationship, we can write

$$\forall x, y \;\; Sibling(x, y) \Leftrightarrow Sibling(y, x) \; .$$

In other cases we will have mixtures. "Everybody loves somebody" means that for every person, there is someone that person loves:

$$\forall x \;\; \exists y \;\; Loves(x, y) \; .$$

[6] There is a variant of the existential quantifier, usually written $\exists^1$ or $\exists!$, that means "There exists exactly one." The same meaning can be expressed using equality statements, as we show in Section 8.2.

On the other hand, to say "There is someone who is loved by everyone," we write

$$\exists y \ \forall x \ \ Loves(x, y) .$$

The order of quantification is therefore very important. It becomes clearer if we insert parentheses. $\forall x \ (\exists y \ Loves(x, y))$ says that *everyone* has a particular property, namely, the property that somebody loves them. On the other hand, $\exists x \ (\forall y \ Loves(x, y))$ says that *someone* in the world has a particular property, namely the property of being loved by everybody.

Some confusion can arise when two quantifiers are used with the same variable name. Consider the sentence

$$\forall x \ [Crown(x) \vee (\exists x \ \ Brother(Richard, x))] .$$

Here the x in $Brother(Richard, x)$ is *existentially* quantified. The rule is that the variable belongs to the innermost quantifier that mentions it; then it will not be subject to any other quantification.[7] Another way to think of it is this: $\exists x \ Brother(Richard, x)$ is a sentence about Richard (that he has a brother), not about x; so putting a $\forall x$ outside it has no effect. It could equally well have been written $\exists z \ Brother(Richard, z)$. Because this can be a source of confusion, we will always use different variables.

Connections between $\forall$ and $\exists$

The two quantifiers are actually intimately connected with each other, through negation. Asserting that everyone dislikes parsnips is the same as asserting there does not exist someone who likes them, and vice versa:

$$\forall x \ \neg Likes(x, Parsnips) \quad \text{is equivalent to} \quad \neg \exists x \ \ Likes(x, Parsnips) .$$

We can go one step further: "Everyone likes ice cream" means that there is no one who does not like ice cream:

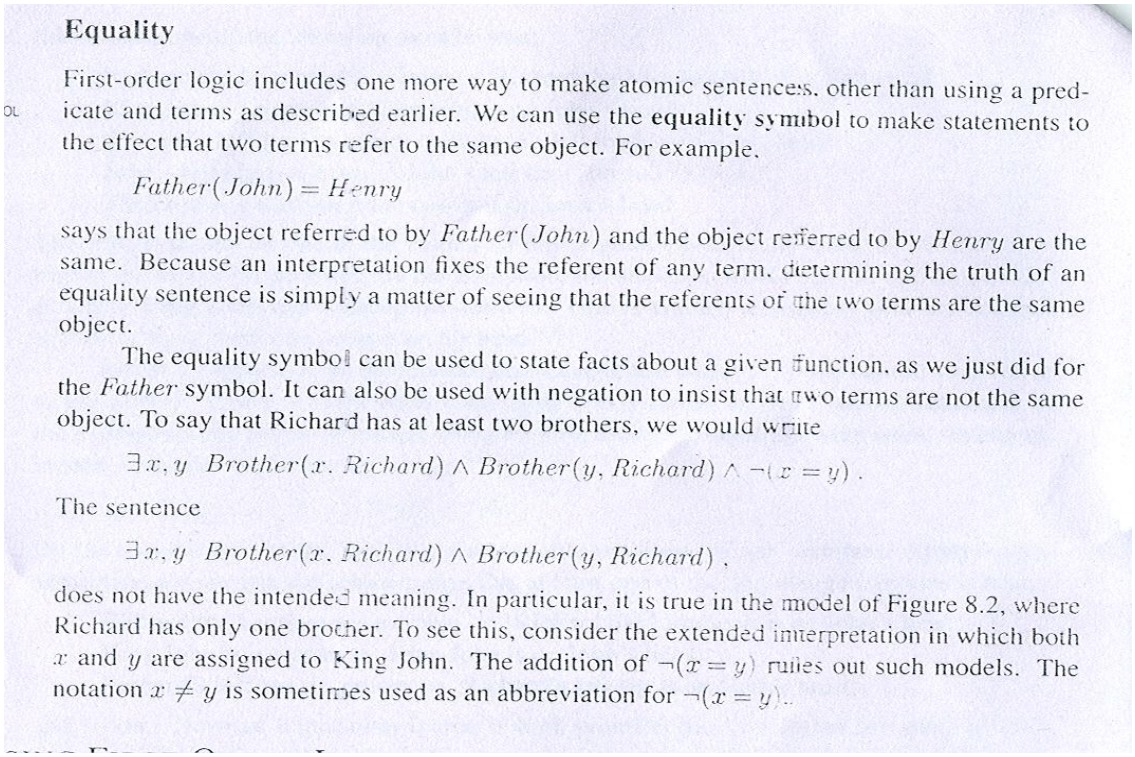
$$\forall x \ \ Likes(x, IceCream) \quad \text{is equivalent to} \quad \neg \exists x \ \neg Likes(x, IceCream) .$$

Because $\forall$ is really a conjunction over the universe of objects and $\exists$ is a disjunction, it should not be surprising that they obey de Morgan's rules. The de Morgan rules for quantified and unquantified sentences are as follows:

$$\begin{array}{lll} \forall x \ \neg P \equiv \neg \exists x \ P & & \neg P \wedge \neg Q \equiv \neg (P \vee Q) \\ \neg \forall x \ P \equiv \exists x \ \neg P & & \neg (P \wedge Q) \equiv \neg P \vee \neg Q \\ \forall x \ P \equiv \neg \exists x \ \neg P & & P \wedge Q \equiv \neg (\neg P \vee \neg Q) \\ \exists x \ P \equiv \neg \forall x \ \neg P & & P \vee Q \equiv \neg (\neg P \wedge \neg Q) . \end{array}$$

Thus, we do not really need both $\forall$ and $\exists$, just as we do not really need both $\wedge$ and $\vee$. Still, readability is more important than parsimony, so we will keep both of the quantifiers.

[7] It is the potential for interference between quantifiers using the same variable name that motivates the slightly baroque mechanism of extended interpretations in the semantics of quantified sentences. The more intuitively obvious approach of substituting objects for every occurrence of x fails in our example because the x in $Brother(Richard, x)$ would be "captured" by the substitution. Extended interpretations handle this correctly because the inner quantifier's assignment for x overrides the outer quantifier's.



Equality

First-order logic includes one more way to make atomic sentences, other than using a predicate and terms as described earlier. We can use the **equality symbol** to make statements to the effect that two terms refer to the same object. For example,

$$Father(John) = Henry$$

says that the object referred to by $Father(John)$ and the object referred to by $Henry$ are the same. Because an interpretation fixes the referent of any term, determining the truth of an equality sentence is simply a matter of seeing that the referents of the two terms are the same object.

The equality symbol can be used to state facts about a given function, as we just did for the *Father* symbol. It can also be used with negation to insist that two terms are not the same object. To say that Richard has at least two brothers, we would write

$$\exists x, y \;\; Brother(x, Richard) \wedge Brother(y, Richard) \wedge \neg(x = y) .$$

The sentence

$$\exists x, y \;\; Brother(x, Richard) \wedge Brother(y, Richard) ,$$

does not have the intended meaning. In particular, it is true in the model of Figure 8.2, where Richard has only one brother. To see this, consider the extended interpretation in which both x and y are assigned to King John. The addition of $\neg(x = y)$ rules out such models. The notation $x \neq y$ is sometimes used as an abbreviation for $\neg(x = y)$.

Rules for constructing Wffs

A predicate name followed by a list of variables such as *P*(*x*, *y*), where *P* is a predicate name, and *x* and *y* are variables, is called an **atomic formula**.

Wffs are constructed using the following rules:

1. *True* and *False* are wffs.
2. Each propositional constant (i.e. specific proposition), and each propositional variable (i.e. a variable representing propositions) are wffs.
3. Each atomic formula (i.e. a specific predicate with variables) is a wff.
4. If *A*, *B*, and *C* are wffs, then so are ¬*A*, (*A* ∧*B*), (*A* ∨*B*), (*A* →*B*), and (*A* ↔*B*).
5. If *x* is a variable (representing objects of the universe of discourse), and *A* is a wff, then so are ∀*x A* and ∃*x A* .

More generally, arguments of predicates are something called a term. Also variables representing predicate names (called predicate variables) with a list of variables can form atomic formulas.

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.

2. Marcus was a Pompeian.

Pompeian(Marcus)

3. All Pompeians were Romans.

$$\forall x \text{ Pompeian}(x) \rightarrow \text{Roman}(x)$$

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.

$$\forall x \text{ Roman}(x) \rightarrow \text{loyalto}(x, \text{Caesar}) \vee \text{hate}(x, \text{Caesar})$$

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

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

$$\forall x\ \text{Roman}(x) \rightarrow ((\text{loyalto}(x,\text{Caesar}) \vee \text{hate}(x,\text{Caesar})) \wedge \sim(\text{loyalto}(x, \text{Caesar}) \wedge \text{hate}(x,\text{Caesar}))$$

6. Everyone is loyal to someone.

$$\forall x \exists y\ \text{loyalto}(x,y)$$

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. 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.

$$\forall x \forall y\ \text{person}(x) \wedge \text{ruler}(y) \wedge \text{tryassassinate}(x,y) \rightarrow \sim\text{loyalto}(x,y)$$

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, 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.

$$\text{tryassassinate}(\text{Marcus}, \text{Caesar})$$

From this brief attempt to convert English sentences into logical statements, it should be clear how difficult the task is.

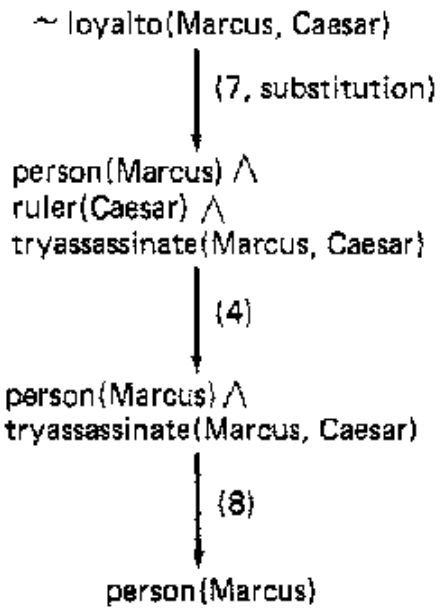


Figure 8: An Attempt to Prove -- ¬Loyalto (Marcus, Caesar)

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:

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 when there are alternative ways of satisfying individual goals. Here, for simplicity, we will show only a single path. Figure 8 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 (Marcus) 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:

9. All men are people.

$$\forall x \ man(x) \rightarrow person(x)$$

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 ways of representing the knowledge (as discussed in connection with 1 and 7 above). Simple representations are desirable but they may preclude certain kinds of reasoning.

• 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.

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)

or

~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 to 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 non theorem.

Another thing it could do is simply to 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.

- **Inference rules of predicate logic**
  - universal instantiation
  - universal generalization

- existential instantiation
- existential generalization
- negation of quatified statement

Predicate logic is more powerful than propositional logic. It allows one to reason about properties and relationships of individual objects. In predicate logic, one can use some additional **inference rules**, which are discussed below, as well as those for propositional logic such as the **equivalences, implications and inference rules**. The following four rules describe when and how the universal and existential quantifiers can be added to or deleted from an assertion.

1. **Universal Instantiation:**

   $\forall x\, P(x)$
   ———————
   $P(c)$

   where *c* is some arbitrary element of the universe.
   Go to **Universal Instantiation** for further explanations and examples.

2. **Universal Generalization:**

   $P(c)$
   ———————
   $\forall x\, P(x)$

   where *P(c)* holds for every element c of the universe of discourse.
   Go to **Universal Generalization** for further explanations and examples.

3. **Existential Instantiation:**

   $\exists x\, P(x)$
   ———————
   $P(c)$

   where *c* is some element of the universe of discourse. It is not arbitrary but must be one for which P(c) is true.
   Go to **Existential Instantiation** for further explanations and examples.

4. **Existential Generalization:**

   $P(c)$
   ———————
   $\exists x\, P(x)$

   where *c* is an element of the universe.
   Go to **Existential Generalization** for further explanations and examples.

Augmenting the representation with computable functions and predicates

In the example we explored in the last section, all of 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

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 shall 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 to gt the arguments 5 and 1. 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 will 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.

$$\forall x \forall t1 \forall t2 \; mortal(x) \wedge born(x,t1) \wedge gt(t2\text{-}t1,150) \rightarrow dead(x,t2)$$

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 1983.

now = 1983

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:

1. Alive means not dead.

$$\forall x \forall t \; alive(x,t) \leftrightarrow \sim dead(x,t)$$

This is not strictly correct, since --~dead implies alive only for animate objects. (Chairs can be both not dead and not alive, or neither.) 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.

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

$$\forall x \forall t1 \forall t2 \; died(x,t1) \wedge gt(t2,t1) \rightarrow dead(x,t2)$$

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. 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(yearl, monthl) if one died during (yearl,month2) and month2 precedes monthl." 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 of the facts we have now represented is given in Figure 9.

Now let's attempt to answer the question "Is Marcus alive?" by proving

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

Two such proofs are shown in Figures 10 and 11. 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 sub goals. In one sense they are,

1. man(Marcus)
2. Pompeian(Marcus)
3. born(Marcus,40)
4. ∀x man(x) → mortal(x)
5. ∀x Pompeian(x) → died(x,79)
6. erupted(volcano,79)
7. ∀x∀t1∀t2 mortal(x) ∧ born(x,t1) ∧ gt(t2−t1,150) → dead(x,t2)
8. now=1983
9. ∀x∀t alive(x,t) ↔ ~dead(x,t)
10. ∀x∀t1∀t2 died(x,t1) ∧ gt(t2,t1) → dead(x,t2)

Figure 9: A Set of Facts about Marcus

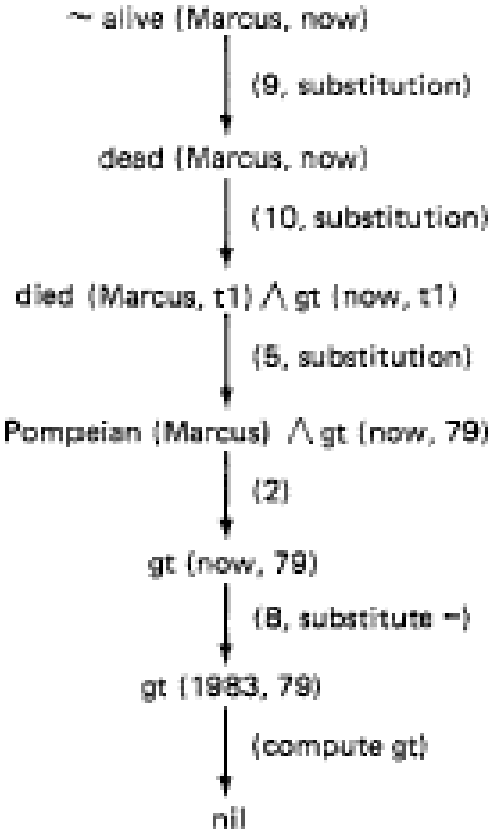


Figure 10: One Way of Proving That Marcus Is Dead

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 Figure 11 look at the step justified by statement 3. We can satisfy the goal

born(Marcus,t1)

using statement 3 by binding t l to 40, but then we must also bind tl in

gt(now-t1,150)

to 40, since the two t l's were the same variable in statement 4, from which the two goals came.

A good computational proof procedure will have 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 of those things will be discussed below. 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 at things. 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 will 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.

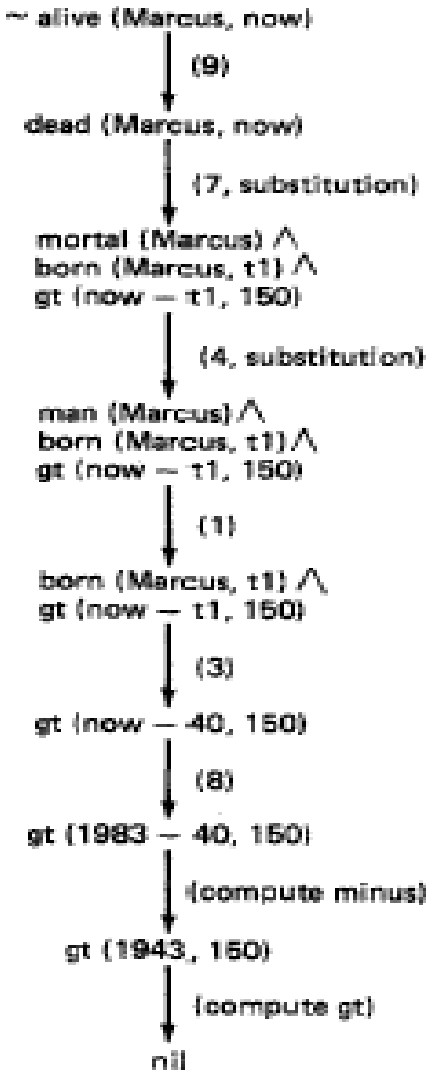


Figure 11: Another Way of Proving That Marcus Is Dead

Resolution

As we suggested above, it would be useful from a computational point of view if we had a proof procedure that carried out in a single operation the variety of processes involved in reasoning with statements in predicate logic. Resolution is such a procedure, which gains its efficiency from the fact that it operates on statements that have been converted to a very convenient standard form, which will be described below.

Resolution produces proofs by *refutation.* In other words, to prove a statement (i.e., show that it is valid), resolution attempts to show that the negation of the statement produces a contradiction with the known statements (i.e., that it is unsatisfiable). This approach contrasts with the technique that we have been using to generate proofs by chaining backward from the theorem to be proved to the axioms. Further discussion of how resolution operates will be much more straightforward after we have discussed the standard form in which statements will be represented, so we will defer it until then.

Other approaches to inference use syntactic operations on sentences, often expressed in standardized forms

Conjunctive Normal Form (CNF—universal)
conjunction of *disjunctions* of *literals*
clauses
E.g., $(A \vee \neg B) \wedge (B \vee \neg C \vee \neg D)$

"product of sums of simple variables or negated simple variables"

Disjunctive Normal Form (DNF—universal)
disjunction of *conjunctions* of *literals*
terms
E.g., $(A \wedge B) \vee (A \wedge \neg C) \vee (A \wedge \neg D) \vee (\neg B \wedge \neg C) \vee (\neg B \wedge \neg D)$

"sum of products of simple variables or negated simple variables"

Horn Form (restricted)
conjunction of *Horn clauses* (clauses with ≤ 1 positive literal)
E.g., $(A \vee \neg B) \wedge (B \vee \neg C \vee \neg D)$
Often written as set of implications:
$B \Rightarrow A$ and $(C \wedge D) \Rightarrow B$

Conversion to Clause Form

Suppose we know that all Romans who know Marcus either hate Caesar or think that anyone who hates anyone is crazy. We could represent that in the following wff:

$$\forall x\ [\text{Roman}(x) \wedge \text{know}(x,\text{Marcus})] \rightarrow [\text{hate}(x,\text{Caesar}) \vee (\forall y\ (\exists z\ \text{hate}(y,z)) \rightarrow \text{thinkcrazy}(x,y))]$$

To use this formula in a proof requires a complex matching process. Then, having matched one piece of it, such as thinkcrazy(x,y), it is necessary to do the right thing with the rest of the formula including the pieces in which the matched part is embedded and those in which it is not. If the formula were in a simpler form, this process would be much easier. The formula would be easier to work with if

• It were flatter, i.e., there was less embedding of components.

• The quantifiers were separated from the rest of the formula so that they did

not need to be considered.

Conjunctive normal form has both of these properties. For example, the formula given above for the feelings of Romans who know Marcus would be represented in conjunctive normal form as

$$\sim\text{Roman}(x) \vee \sim\text{know}(x,\text{Marcus}) \vee \text{hate}(x,\text{Caesar}) \vee \sim\text{hate}(y,z) \vee \text{thinkcrazy}(x,z)$$

Since there exists an algorithm for converting any wff into conjunctive normal form, we lose no generality if we employ a proof procedure (such as resolution) that operates only on wff's in this form. To convert a wff into conjunctive normal form, perform the following sequence of steps:

1. Eliminate $\rightarrow$, using the fact that $a \rightarrow b$ is equivalent to $\sim a \vee b$. Performing this transformation on the wff given above yields

$$\forall x \sim[\text{Roman}(x) \wedge \text{know}(x,\text{Marcus})] \vee [\text{hate}(x,\text{Caesar}) \vee (\forall y \sim(\exists z\ \text{hate}(y,z)) \vee \text{thinkcrazy}(x,y))]$$

2. Reduce the scope of $\sim$, using the fact that $\sim(\sim p)=p$, deMorgan's laws [which say that $\sim(a \wedge b)=\sim a \vee \sim b$ and $\sim(a \vee b)=\sim a \wedge \sim b$] and the standard correspondences between quantifiers [$\sim\forall x P(x)=\exists x \sim P(x)$ and $\sim\exists x P(x)=\forall x \sim P(x)$]. Performing this transformation on the wff from step 1 yields

$$\forall x[\sim\text{Roman}(x) \vee \sim\text{know}(x,\text{Marcus})] \vee [\text{hate}(x,\text{Caesar}) \vee (\forall y \forall z \sim\text{hate}(y,z) \vee \text{thinkcrazy}(x,y))]$$

3. Standardize variables so that each quantifier binds a unique variable. Since variables are just dummy names, this process cannot affect the truth value of the wff. For example, the formula

$$\forall x P(x) \vee \forall x Q(x)$$

would be converted to

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

This step is in preparation for the next.

4. Move all quantifiers to the left of the formula, without changing their relative order. This is possible since there is no conflict among variable names. Performing this operation on the formula of step 2, we get

$$\forall x \forall y \forall z\ [\sim\text{Roman}(x) \vee \sim\text{know}(x,\text{Marcus})] \vee [\text{hate}(x,\text{Caesar}) \vee (\sim\text{hate}(y,z) \vee \text{thinkcrazy}(x,y))]$$

At this point, the formula is in what is known as *prenex normal form*. It consists of a *prefix* of quantifiers followed by a *matrix*, which is quantifier-free.

5. Eliminate existential quantifiers. A formula that contains an existentially quantified variable asserts that there is a value that can be substituted for

the variable that makes the formula true. We can eliminate the quantifier by substituting for the variable a reference to a function that produces the desired value. Since we do not necessarily know how to produce the value, we must create a new function name for every such replacement. We make no assertions about these functions except that they must exist. So, for example, the formula

$\exists y$ President(y)

can be transformed into the formula

President(S1)

where S1 is a function of no arguments that somehow produces a value that satisfies President.

If existential quantifiers occur within the scope of universal quantifiers, then the value that satisfies the predicate may depend on the values of the universally quantified variables. For example, in the formula

$\forall x \exists y$ fatherof(y,x)

the value of y that satisfies fatherof depends on the particular value of x. Thus we must generate functions with the same number of arguments as the number of universal quantifiers in whose scope the expression occurs. So this example would be transformed into

$\forall x$ fatherof(S2(x),x)

These generated functions are called *Skolem functions*. Sometimes ones with no arguments are called *Skolem constants*.

6. Drop the prefix. At this point, all remaining variables are universally quantified, so the prefix can just be dropped and any proof procedure we use can simply assume that any variable it sees is universally quantified. Now the formula produced in step 4 appears as

$$[\sim \text{Roman}(x) \vee \sim \text{know}(x,\text{Marcus})] \vee [\text{hate}(x,\text{Caesar}) \vee (\sim \text{hate}(y,z) \vee \text{thinkcrazy}(x,y))]$$

7. Convert the matrix into a conjunction of disjuncts. In the case of our example, since there are no AND's, all that is necessary to do is to exploit the associative property of OR [i.e., $a \vee (b \vee c) = (a \vee b) \vee c$] and simply remove the parentheses, giving

$$\sim Roman(x) \vee \sim know(x,Marcus) \vee hate(x,Caesar) \vee \sim hate(y,z) \vee thinkcrazy(x,y)$$

However, it is also frequently necessary to exploit the distributive property [$(a \wedge b) \vee c = (a \vee c) \wedge (b \vee c)$]. For example, the formula

$$(winter \wedge wearingboots) \vee (summer \wedge wearingsandals)$$

becomes, after one application of the rule

$$[winter \vee (summer \wedge wearingsandals)] \wedge [wearingboots \vee (summer \wedge wearingsandals)]$$

and then, after a second application, required since there are still conjuncts joined by OR's

$$(winter \vee summer) \wedge (winter \vee wearingsandals) \wedge (wearingboots \vee summer) \wedge (wearingboots \vee wearingsandals)$$

8. Call each conjunct a separate *clause*. In order for a wff to be true, all the clauses that are generated from it must be true. If we are going to be working with several wff's, all the clauses generated by each of them can now be combined to represent the same set of facts as were represented by the original wff's.

9. Standardize apart the variables in the set of clauses generated in step 8. By this we mean rename the variables so that no two clauses make reference to the same variable. In making this transformation, we rely on the fact that

$$(\forall x P(x) \wedge Q(x)) = \forall x P(x) \wedge \forall x Q(x)$$

Thus since each clause is a separate conjunct and all the variables are universally quantified, there need be no relationship between the variables of two clauses, even if they were generated from the same wff.

Performing this final step of standardization is important because during the resolution procedure it will sometimes be necessary to instantiate a universally quantified variable (i.e., substitute for it a particular value). But in general, we will want to keep

clauses in their most general form as long as possible. So when a variable is instantiated we want to know the minimum number of substitutions that must be made to preserve the truth value of the system.

After applying this entire procedure to a set of wff's, we will have a set of clauses, each of which is a disjunction of *literals.* These clauses can now be exploited by the resolution procedure to generate proofs.

The Basis of Resolution

The resolution procedure is a simple iterative process, at each step of which two clauses, called the *parent clauses,* are compared *(resolved),* yielding a new clause that has been inferred from them. The new clause represents ways that the two parent clauses interact with each other. Suppose that there are two clauses in the system:

winter $\vee$ summer
$\neg$winter $\vee$ cold

Recall that this means that both clauses must be true (i.e., the clauses, although they look independent, are really conjoined). Now we observe that precisely one of winter and ¬winter will be true at any point. If winter is true, then cold must be true to guarantee the truth of the second clause. If ¬winter is true, then summer must be true to guarantee the truth of the first clause. Thus we see that from these two clauses we can deduce

summer V cold.

This is the deduction that the resolution procedure will make. Resolution operates by taking two clauses that each contain the same literal, in this example, winter. The literal must occur in positive form in one clause and in negative form in the other. The *resolvent* is obtained by combining all of the literals of the two parent clauses except the ones that cancel. If the clause that is produced is the empty clause, then a contradiction has been found. For example, the two clauses

winter , ¬winter

will produce the empty clause. If a contradiction exists, then eventually it will be found. Of course, if no contradiction exists, it is possible that the procedure will never terminate, although as we will see, there are often ways of detecting that no contradiction exists. So far, we have discussed only resolution in propositional logic. In predicate logic, the situation is more complicated since we must consider all possible ways of substituting values for the variables.

Resolution in Propositional Logic

In order to make it clear how resolution works, we will first present the resolution procedure for propositional logic. We will then expand it to include predicate logic. In propositional logic, the procedure for producing a proof by resolution of proposition S with respect to a set of axioms F is the following:

1. Convert all the propositions of F to clause form.
2. Negate S 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 ¬L, such that one of the parent clauses contains L and the other contains ¬L , then eliminate both L and ¬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.

Given Axioms	Converted to Clause Form	
p	p	1.
(p ∧ q) → r	~p ∨ ~q ∨ r	2.
(s ∨ t) → q	~s ∨ q	3.
	~t ∨ q	4.
t	t	5.

Figure 12: A Few Facts in Propositional Logic

Let's look at a simple example. Suppose we are given the axioms shown in the first column of Figure 12 and we want to prove R. First we convert the axioms to clause form, as shown in the second column of the figure. Then we negate R, producing ¬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.

We might, for example, generate the sequence of resolvents shown in Figure 13. We begin by resolving with the clause ¬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 all of which are assumed to be true. It generates new clauses that represent restrictions on the way each of those original clauses can be made true, based on information provided by the others. 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.

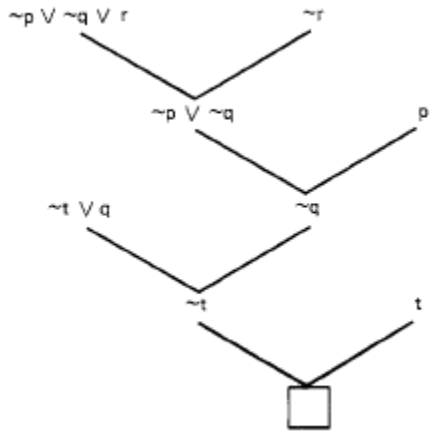


Figure 13: Resolution in Propositional Logic

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:- ~p , ~q, or r. But we are assuming that ~r is true. Given that, the only way for proposition 2 to be true is for one of two things to be true: - ~p or ~q. That is what the first resolvent clause says. But proposition 1 says that p is true, which means that , ~p cannot be true, which leaves only one way for proposition 2 to be true, namely for ~q to be true (as shown in the second resolvent clause). Proposition 4 can be true if either ~t or q is true. But since we now know that ~q must be true, the only way for proposition 4 to be true is for ~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).

Unification Algorithm

In propositional logic, it is easy to determine that two literals cannot both be true at the same time. Simply look for L and ~L. In predicate logic, this matching process is more complicated, since bindings of variables must be considered. For example, man(Henry) and ~man(Henry) is a contradiction, while man(Henry) and ~man(Spot) is not. Thus, in order to determine contradictions, we need a matching procedure that compares two literals and discovers whether there exists a set of substitutions that makes them identical. There is a straightforward recursive procedure, called the *unification algorithm* that does just this. The basic idea of unification is very simple. It is easiest to describe if we represent each literal as a list, where the first element is the name of the predicate and the remaining elements are the arguments, each of which is either a single element (which we will call an atom, following LISP terminology) or it is another list. So we might have literals such as

(tryassassinate Marcus Caesar)

(tryassassinate Marcus (rulerof Rome))

To attempt to unify two literals, we first check to see if their first elements are the same. If so, we can proceed. Otherwise, there is no way they can be unified, regardless of their arguments. For example, the two literals

(tryassassinate Marcus Caesar)

(hate Marcus Caesar)

cannot be unified. If the first elements match, then we must check the remaining elements, one pair at a time. If the first matches, we can continue with the second, and so on. To test each of the remaining elements, we can simply call the unification procedure recursively. The matching rules are simple. Different constants, functions, or predicates cannot match; identical ones can. A variable can match another variable, any constant, or a function or predicate expression with the restriction that the function or predicate expression must not contain any instances of the variable being matched.

The only complication in this procedure is that we must find a single, consistent substitution for the entire literal, not separate ones for each piece of it. To do this, we must take each substitution that we find and apply it to the remainder of the literals before we continue trying to unify them. For example, suppose we want to unify the expressions

P(x, x)

P(y, z)

The two instances of P match fine. Next we compare x and y, and decide that if we substitute y for x, they could match. We will write *that* substitution as

y/x

(We could, of course, have decided instead to substitute x for y, since they are both just dummy variable names. The algorithm will simply pick one of these two substitutions.) But now, if we simply continue and match x and z, we produce the substitution z/x. But we cannot substitute both y and z for x, so we have not produced a consistent substitution. What we need to do is, after finding the first substitution y/x, to make that substitution in the remaining pieces of the literals, giving

(y)

(z)

Now we can attempt to unify these literals, which succeeds with the substitution z/y. The entire unification process has now succeeded with a substitution that is the composition of the two substitutions we found. We write the composition as

(z/y)(y/x)

following standard notation for function composition. In general, the substitution (al/a2, a3/a4 ....)(bl/b2, b3/b4 ....)... means to apply all the substitutions of the rightmost list, then take the result and apply all the ones of the next list, and so forth, until all substitutions have been applied. The object of the unification procedure is to discover at least one substitution that causes two literals to match. Usually, if there is one such substitution there are many. For example, the literals

hate(x,y)

hate(Marcus,z)

could be unified with any of the following substitutions:

(Marcus/x,z/y)

(Marcus/x,y/z)

(Marcus/x, Caesar/y, Caesar/z)

(Marcus/x,Polonius/y,Polonius/z)

The first two of these are equivalent except for lexical variation. But the second two, although they produce a match, also produce a substitution that is more restrictive than absolutely necessary for the match. Because the final substitution produced by the unification process will be used by the resolution procedure, it is useful to generate the most general unifier possible. The algorithm shown below will do that. Having explained the operation of the unification algorithm, we can now state it concisely. We describe a procedure UNIFY(L1 ,L2), which returns as its value a list representing the composition of the substitutions that were performed during the match. The empty list, NIL, indicates that a match was found without any substitutions. The list consisting of the single value F indicates that the unification procedure failed.

Unify(L1,L2)

1. if L1 or L2 is an atom then do
    1. if L1 and L2 are identical then return NIL
    2. else if L1 is a variable then do
        1. if L1 occurs in L2 then return F, else return (L2/L1)
    3. else if L2 is a variable then do
        1. if L2 occurs in L1 then return F, else return (L1/L2)

    else return F
2. if length(L1) is not equal length(L2) then return F
3. set SUBST to NIL (At the end of this procedure, SUBST will contain all the substitutions used to unify L1 and L2.)

4. for i := 1 to number of elements in L1 do
    1. call unify with the i'th element of L1 and the i'th element of L2, putting result in S.
    2. if S = F then return F.
    3. if S is not equal to NIL then do
        1. apply S to the remainder of both L1 and L2
        2. SUBST := APPEND(S,SUBST)

return SUBST

The only part of this algorithm that we have not yet discussed is the check, in steps 1.2 and 1.3, to make sure that an expression involving a given variable is not unified with that variable. Suppose we were attempting to unify the expressions

(f x x)

(f g(x) g(x))

If we accepted g(x) as a substitution for x, then we would have to substitute it for x in the remainder of the expressions. But this leads to infinite recursion, since it will never be possible to eliminate x.

Resolution in Predicate Logic

We now have an easy way of determining that two literals are contradictory if one of them can be unified with the not of the other. So, for example, man(x) and ~man(spot) are contradictory, since man(x) and man(Spot) can be unified. This corresponds to the intuition that says that it can not be true of all x that man(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 shall use the unification algorithm to locate pairs of literals that cancel out. We will 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. ~man(x1) \/ mortal(x1)

The literal man(Marcus) can be unified with the literal man(x l) with the substitution Marcus/xl, telling us that for xl = Marcus, ~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(xl). Clause 2 says that for a given xl, either ~man(x l) or mortal(x1). So for it to be true we can now conclude only that mortal(Marcus)must be true. It is not necessary that mortal(xl) be true for all xl, since for some values of xl ~man(xl) might be true, making mortal(xl) 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 if 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 been 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 S:

1. Convert all the statements of F to clause form.
2. Negate S and convert the result to clause form. Add it to the set of clauses obtained in 1.
3. Repeat until either a contradiction is found, no progress can be made, or a predetermined amount of effort has been expended:
    1. Select two clauses. Call these the parent clauses.
    2. Resolve them together. The resolvent will be the disjunction of all of the literals of both of the parent clauses with appropriate substitutions performed and with the following exception: If there is a pair of literals T1 and ~T2 such that one of the parent clauses contains T1 and the other contains T2 and if T1 and T2 are unifiable, then neither T1 nor T2 should appear in the resolvent. We will call T1 and T2 *complimentary literals*. Use the substitution produced by the unification to create the resolvent.
    3. 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 or not the predicate is negated. Then, given a particular clause, possible resolvents that contain a complimentary occurrence of one of its predicates can be located directly.

• Eliminate certain clauses as soon as they are generated so that they cannot participate in later resolutions. Two kinds of clauses should be eliminated: tautologies (which can never be unsatisfiable) and clauses that are subsumed by other clauses (i.e., they are easier to satisfy. For example, p V 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 with 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 us 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 5.2. To use them in resolution proofs, they must be converted to clause form as described in Section 5.4.i. Figure 5-9(a) shows the results of doing that conversion. Figure 5-9(b) shows a resolution proof of the statement

hate(Marcus, Caesar)

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

~hate(Marcus,Caesar)

To do so, we would have added

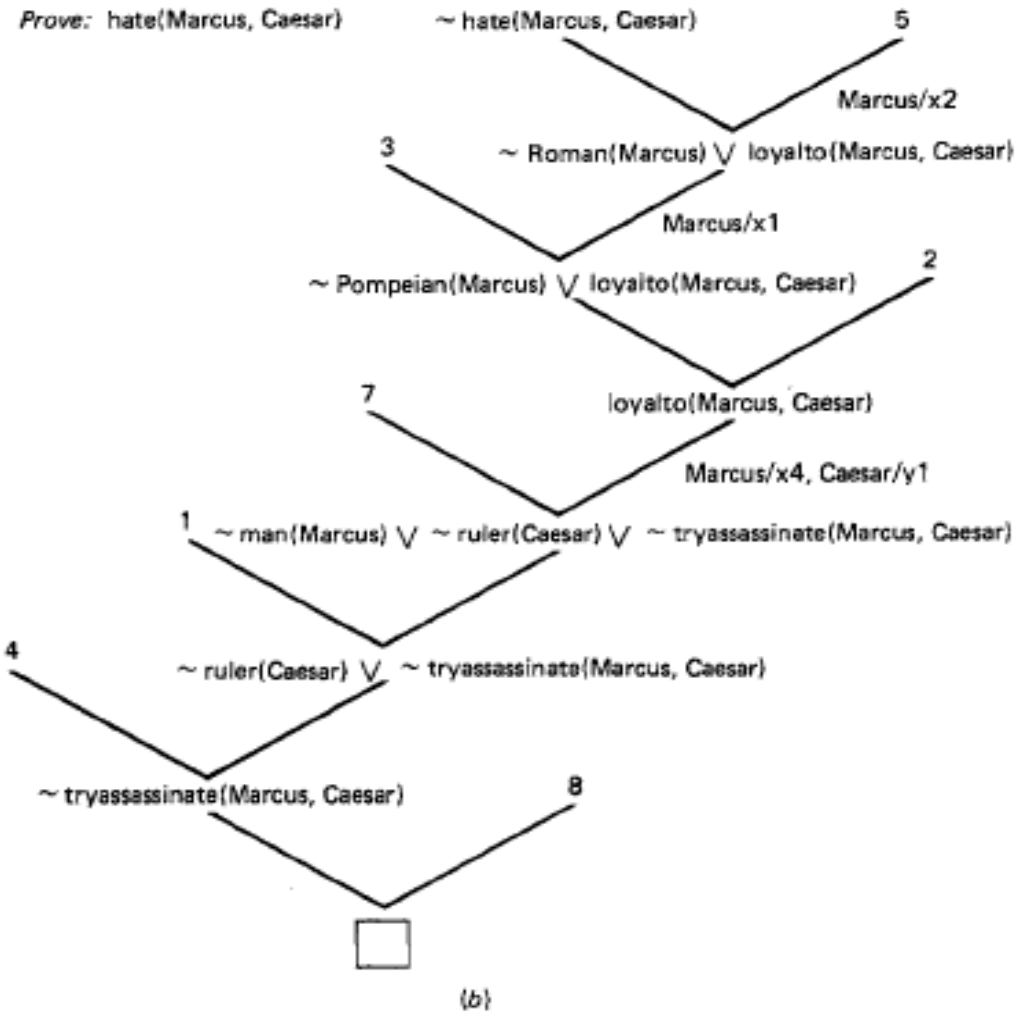
hate(Marcus, Caesar)

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 ~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 partway through it, as shown in the example in Figure 5-10(a), based on the axioms given in Figure 5-9.

Axioms in clause form:

1. man(Marcus)
2. Pompeian(Marcus)
3. ~ Pompeian(x1) ∨ Roman(x1)
4. ruler(Caesar)
5. ~ Roman(x2) ∨ loyalto(x2, Caesar) ∨ hate(x2, Caesar)
6. loyalto(x3, f1(x3))
7. ~ man(x4) ∨ ~ ruler(y1) ∨ ~ tryassassinate(x4, y1) ∨ ~ loyalto(x4, y1)
8. tryassassinate(Marcus, Caesar)

(a)



(b)

Figure 5.9: A Resolution Proof

But suppose our knowledge base contained the two additional statements
(9) persecute(x,y)→hate(y,x)
(10) hate(x,y)→persecute(y,x)
Converting to clause form, we get
(9) ~persecute(x5,y2) ∨ hate(y2,x5)
(10) ~hate(x6,y3) ∨ persecute(y3,x6)
These statements enable the proof of Figure 5-10(a) to continue as shown

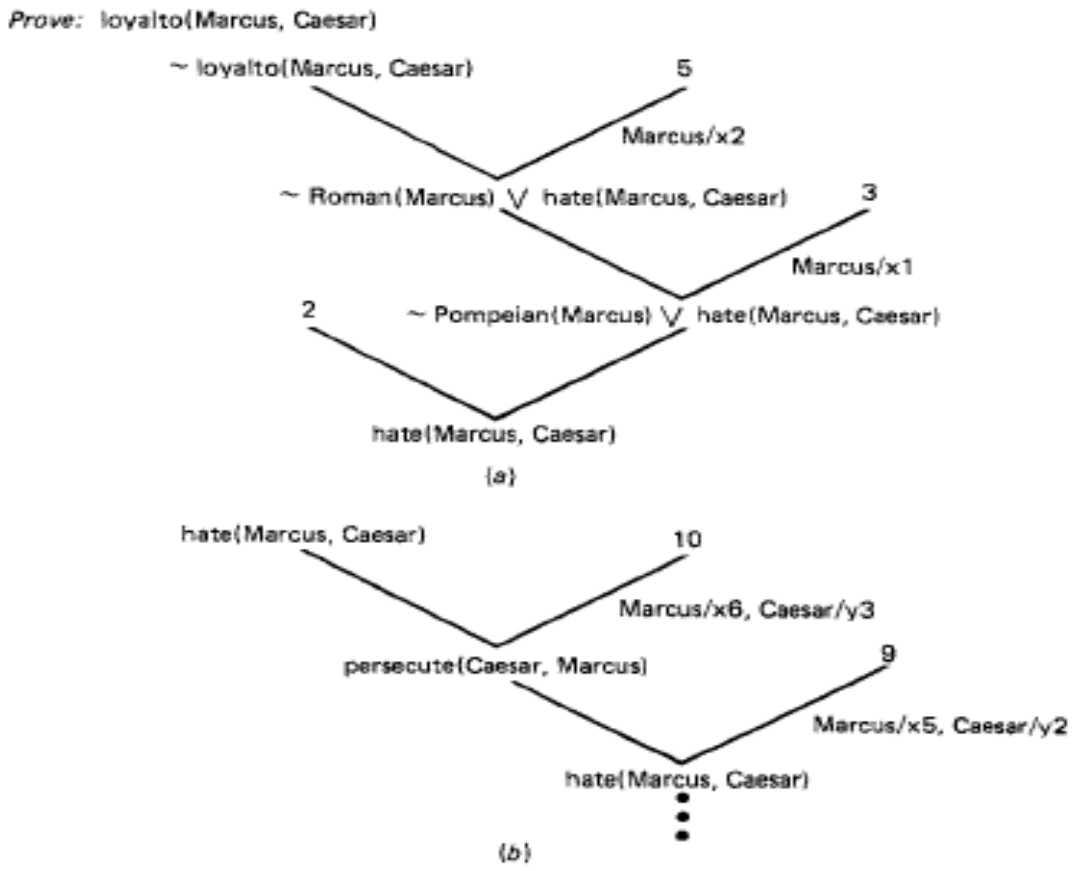


Figure 5.10: An unsuccessful attempt at Resolution

in Figure 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. Recall that the final step of the process of converting a set of formulas to clause from 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

~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

~mother(a,b) ∨ woman(a)

In its pure form, resolution requires all the knowledge it uses to be represented in the form of clauses. But as was pointed out in Section 5.3, it is often more efficient to represent certain kinds of information in the form of com

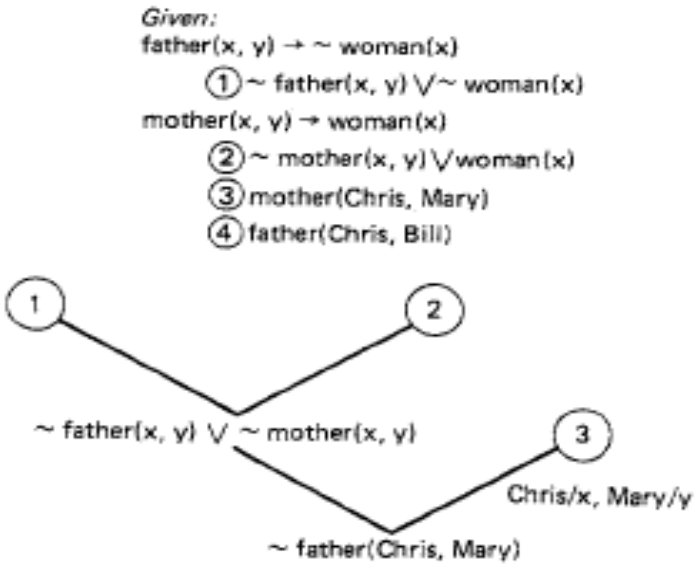


Figure 5.11: The Need to Standardize variables

putable 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

~ 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 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.

The Need to Try Several Substitutions

Resolution provides a very good way of finding a refutation proof without actually trying all of the substitutions that Herbrand's theorem suggests might be necessary. But it does not always eliminate the necessity of trying more than one substitution. For example, suppose we know, in addition to the statements in Section 5.2, that

hate(Marcus ,Paulus)
hate (M arcus ,Julian)

Now if we want to prove that Marcus hates some ruler, we would be likely to try each of the substitutions shown in Figure 5-13(a) and (b) before finding the contradiction shown in (c). Sometimes there is no way short of very good luck to avoid trying several substitutions.

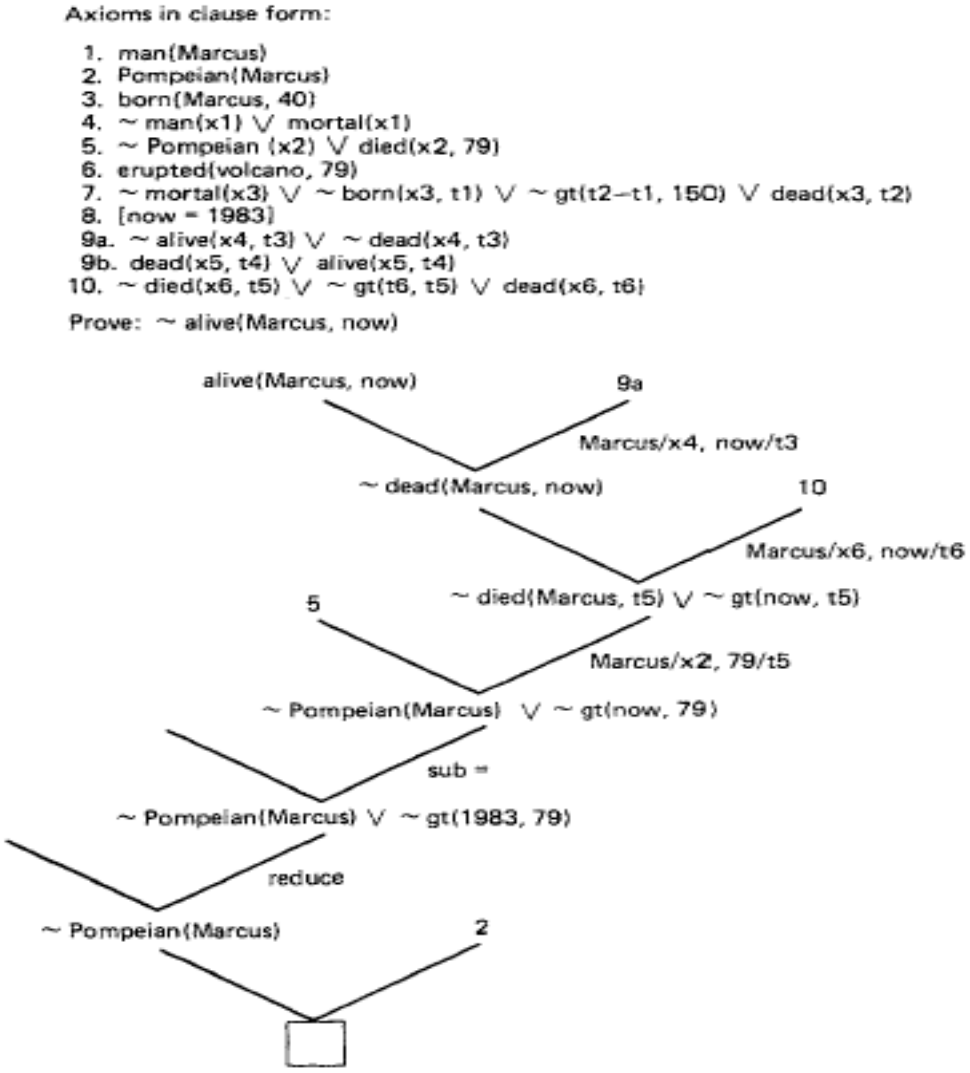


Figure 5-12: Using Resolution with Equality and Reduce

Question Answering

Very early in the history of A.I. it was realized that theorem-proving techniques could be applied to the problem of answering questions. As we have already suggested, this seems natural since both deriving theorems from axioms and deriving new facts (answers) from old facts employ the process of deduction. We have already shown how resolution can be used to answer yes-no questions, such as "Is Marcus alive?" In this section, we will show how resolution can be used to answer fill-in-the-blank questions, such as "When did Marcus die?" or "Who tried to assassinate a ruler?" Answering these questions involves finding a

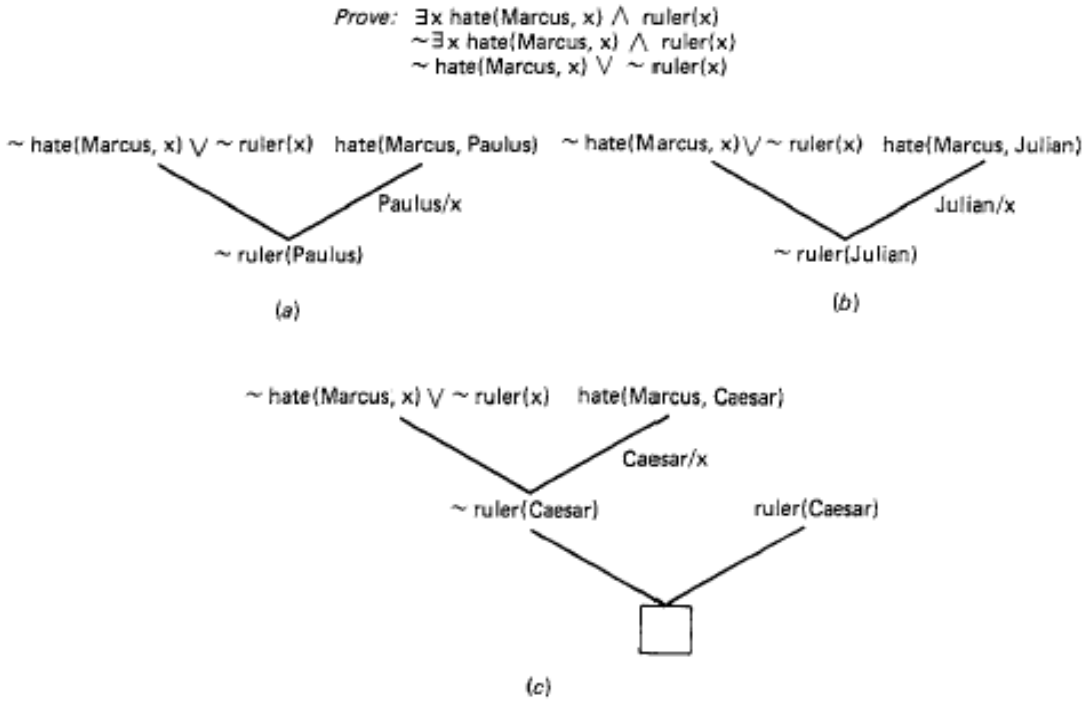


Figure 5-13: Trying Several Substitutions

known statement that matches the terms given in the question and then responding with another piece of that same statement that fills the slot demanded by the question. For example, to answer the question "When did Marcus die?" we need a statement of the form

died(Marcus,??)

with ?? actually filled in by some particular year. So, since we can prove the statement

died(Marcus,79)

we can respond with the answer 79.It turns out that the resolution procedure provides an easy way of locating just the statement we need and finding a proof for it. Let's continue with the example question "When did Marcus die?" In order to be able to answer this question, it must first be true that Marcus died. Thus it must be the case that

$\exists t$ died(Marcus,t)

A reasonable first step then might be to try to prove this. To do so using resolution, we attempt to show that

$\sim\exists t$ died(Marcus,t)

produces a contradiction. What does it mean for that statement to produce a contradiction? Either it conflicts with a statement of the form

∀t died(Marcus,t)

in which case we can either answer the question by reporting that there are many times at which Marcus died, or we can simply pick one such time and respond with it. The other possibility is that we produce a contradiction with one or more specific statements of the form

died(Marcus,date)

Whatever value of date we use in producing that contradiction is the answer we want. The value that proves that there is a value and thus that the statement that there is no such value is inconsistent is exactly the value we want.

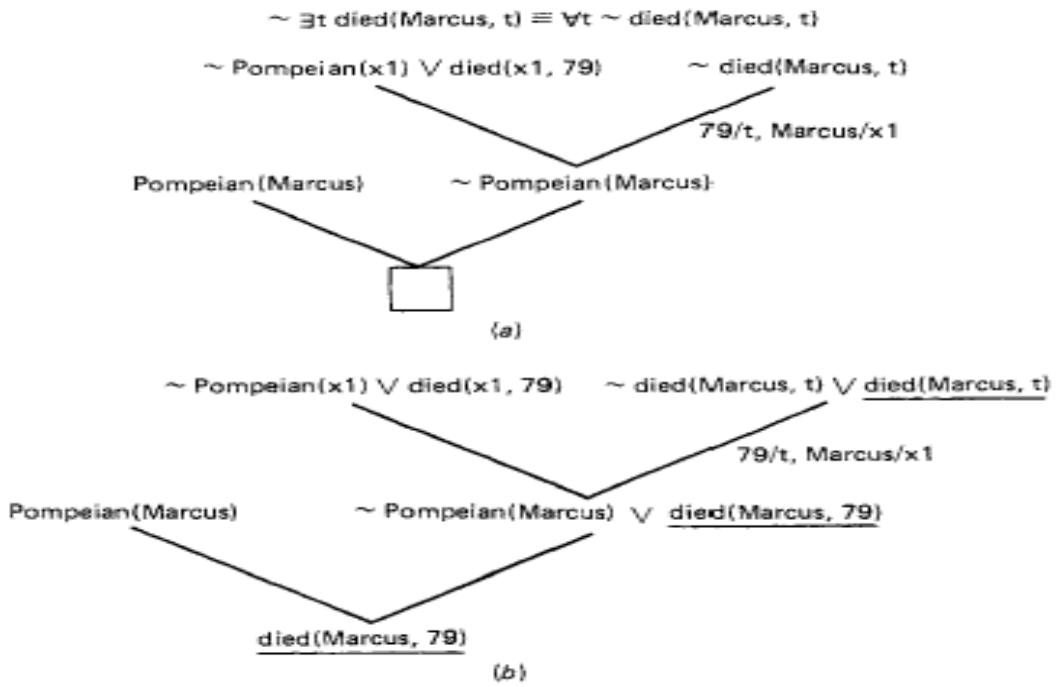


Figure 5-14: Answer Extraction Using Resolution

Figure 5-14(a) shows how the resolution process finds the statement for which we are looking. The answer to the question can then be derived from the chain of unifications that lead back to the starting clause. We can eliminate the necessity for this final step by adding an additional expression to the one we are going to use to try to find a contradiction. This new expression will simply be the one we are trying to prove true (i.e., it will be the negation of the expression that is actually used in the resolution). We can tag it with a special marker so that it will not interfere with the resolution process. (In the figure, it is shown underlined.) It will just get carried along, but each time unification is done, the variables in this dummy expression will be bound just as are the ones in the clauses that are actively being used. Instead of terminating upon reaching the nil clause, the resolution procedure will terminate when all that is left is the dummy expression. The bindings of its variables at that point provide the answer to the question. Figure 5-14(b) shows how this process produces an answer to our question.

Unfortunately, given a particular representation of the facts in a system, there will usually be some questions that cannot be answered using this mechanism. For example, suppose that we want to answer the question "What happened in 79 A.D.?" using the statements in Section 5.3. In order to answer the question, we need to prove that something happened in 79. We need to prove

$\exists x$ event(x,79)

and to discover a value for x. But we do not have any statements of the form

[event(x,y)].

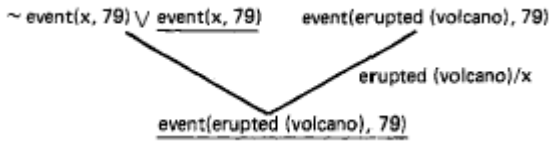


Figure 5-15: Using the New Representation

We can, however, answer the question if we change our representation. Instead of saying

erupted(volcano, 79)

we can say

event(erupted(volcano), 79)

Then the simple proof shown in Figure 5-15 enables us to answer the question. This new representation has the drawback that it is more complex than the old one. And it still does not make it possible to answer all conceivable questions. In general, it is necessary to decide on the kinds of questions that will be asked and to design a representation appropriate for those questions. Of course, yes-no and fill-in-the-blank questions are not the only kinds one could ask. For example, we might ask how to do something. So we have not yet completely solved the problem of question answering. In later chapters, we will discuss some other methods for answering a variety of questions. Some of them exploit resolution; others do not.

A forward-chaining algorithm for propositional definite clauses was given in Section 7.5. The idea is simple: start with the atomic sentences in the knowledge base and apply Modus Ponens in the forward direction, adding new atomic sentences, until no further inferences can be made. Here, we explain how the algorithm is applied to first-order definite clauses and how it can be implemented efficiently. Definite clauses such as $Situation \Rightarrow Response$ are especially useful for systems that make inferences in response to newly arrived information. Many systems can be defined this way, and reasoning with forward chaining can be much more efficient than resolution theorem proving. Therefore it is often worthwhile to try to build a knowledge base using only definite clauses so that the cost of resolution can be avoided.

First-order definite clauses

First-order definite clauses closely resemble propositional definite clauses (page 217): they are disjunctions of literals of which *exactly one is positive*. A definite clause either is atomic or is an implication whose antecedent is a conjunction of positive literals and whose consequent is a single positive literal. The following are first-order definite clauses:

$$King(x) \wedge Greedy(x) \Rightarrow Evil(x) .$$
$$King(John) .$$
$$Greedy(y) .$$

Unlike propositional literals, first-order literals can include variables, in which case those variables are assumed to be universally quantified. (Typically, we omit universal quantifiers when writing definite clauses.) Definite clauses are a suitable normal form for use with Generalized Modus Ponens.

Not every knowledge base can be converted into a set of definite clauses, because of the single-positive-literal restriction, but many can. Consider the following problem:

> The law says that it is a crime for an American to sell weapons to hostile nations. The country Nono, an enemy of America, has some missiles, and all of its missiles were sold to it by Colonel West, who is American.

```
function FOL-FC-ASK(KB, α) returns a substitution or false
  inputs: KB, the knowledge base, a set of first-order definite clauses
          α, the query, an atomic sentence
  local variables: new, the new sentences inferred on each iteration

  repeat until new is empty
      new ← { }
      for each sentence r in KB do
          (p1 ∧ ... ∧ pn ⇒ q) ← STANDARDIZE-APART(r)
          for each θ such that SUBST(θ, p1 ∧ ... ∧ pn) = SUBST(θ, p'1 ∧ ... ∧ p'n)
                      for some p'1, ..., p'n in KB
              q' ← SUBST(θ, q)
              if q' is not a renaming of some sentence already in KB or new then do
                  add q' to new
                  φ ← UNIFY(q', α)
                  if φ is not fail then return φ
      add new to KB
  return false
```

Figure 9.3 A conceptually straightforward, but very inefficient, forward-chaining algorithm. On each iteration, it adds to KB all the atomic sentences that can be inferred in one step from the implication sentences and the atomic sentences already in KB.

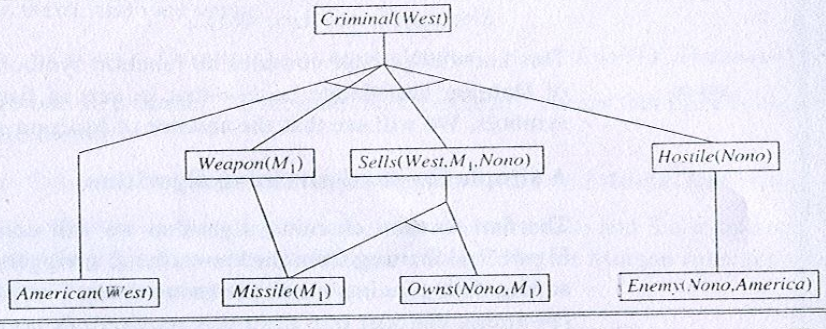


Figure 9.4 The proof tree generated by forward chaining on the crime example. The initial facts appear at the bottom level, facts inferred on the first iteration in the middle level, and facts inferred on the second iteration at the top level.

- On the second iteration, rule (9.3) is satisfied with $\{x/West, y/M_1, z/Nono\}$, and $Criminal(West)$ is added.

Figure 9.4 shows the proof tree that is generated. Notice that no new inferences are possible at this point because every sentence that could be concluded by forward chaining is already contained explicitly in the KB. Such a knowledge base is called a **fixed point** of the inference process. Fixed points reached by forward chaining with first-order definite clauses are similar

to those for propositional forward chaining (page 219); the principal difference is that a first-order fixed point can include universally quantified atomic sentences.

FOL-FC-ASK is easy to analyze. First, it is **sound**, because every inference is just an application of Generalized Modus Ponens, which is sound. Second, it is **complete** for definite clause knowledge bases; that is, it answers every query whose answers are entailed by any knowledge base of definite clauses. For Datalog knowledge bases, which contain no function symbols, the proof of completeness is fairly easy. We begin by counting the number of possible facts that can be added, which determines the maximum number of iterations. Let k be the maximum **arity** (number of arguments) of any predicate, p be the number of predicates, and n be the number of constant symbols. Clearly, there can be no more than pn^k distinct ground facts, so after this many iterations the algorithm must have reached a fixed point. Then we can make an argument very similar to the proof of completeness for propositional forward chaining. (See page 219.) The details of how to make the transition from propositional to first-order completeness are given for the resolution algorithm in Section 9.5.

For general definite clauses with function symbols, FOL-FC-ASK can generate infinitely many new facts, so we need to be more careful. For the case in which an answer to the query sentence q is entailed, we must appeal to Herbrand's theorem to establish that the algorithm will find a proof. (See Section 9.5 for the resolution case.) If the query has no answer, the algorithm could fail to terminate in some cases. For example, if the knowledge base includes the Peano axioms

$$NatNum(0)$$
$$\forall n \;\; NatNum(n) \Rightarrow NatNum(S(n))$$

then forward chaining adds $NatNum(S(0))$, $NatNum(S(S(0)))$, $NatNum(S(S(S(0))))$, and so on. This problem is unavoidable in general. As with general first-order logic, entailment with definite clauses is semidecidable.

Efficient forward chaining

The forward chaining algorithm in Figure 9.3 is designed for ease of understanding rather than for efficiency of operation. There are three possible sources of complexity. First, the "inner loop" of the algorithm involves finding all possible unifiers such that the premise of a rule unifies with a suitable set of facts in the knowledge base. This is often called **pattern matching** and can be very expensive. Second, the algorithm rechecks every rule on every iteration to see whether its premises are satisfied, even if very few additions are made to the knowledge base on each iteration. Finally, the algorithm might generate many facts that are irrelevant to the goal. We will address each of these sources in turn.

Matching rules against known facts

The problem of matching the premise of a rule against the facts in the knowledge base might seem simple enough. For example, suppose we want to apply the rule

$$Missile(x) \Rightarrow Weapon(x) \,.$$

Then we need to find all the facts that unify with $Missile(x)$; in a suitably indexed knowledge base, this can be done in constant time per fact. Now consider a rule such as

$$Missile(x) \wedge Owns(Nono, x) \Rightarrow Sells(West, x, Nono) .$$

Again, we can find all the objects owned by Nono in constant time per object; then, for each object, we could check if whether is a missile. If the knowledge base contains many objects owned by Nono and very few missiles, however, it would be better to find all the missiles first and then check whether they are owned by Nono. This is the **conjunct ordering** problem: find an ordering to solve the conjuncts of the rule premise so that the total cost is minimized. It turns out that finding the optimal ordering is NP-hard, but good heuristics are available. For example, the **most constrained variable** heuristic used for CSPs in Chapter 5 would suggest ordering the conjuncts to look for missiles first if there are fewer missiles than objects that are owned by Nono.

The connection between pattern matching and constraint satisfaction is actually very close. We can view each conjunct as a constraint on the variables that it contains—for example, $Missile(x)$ is a unary constraint on x. Extending this idea, *we can express every finite-domain CSP as a single definite clause together with some associated ground facts.* Consider the map-coloring problem from Figure 5.1, shown again in Figure 9.5(a). An equivalent formulation as a single definite clause is given in Figure 9.5(b). Clearly, the conclusion $Colorable()$ can be inferred only if the CSP has a solution. Because CSPs in general include 3SAT problems as special cases, we can conclude that *matching a definite clause against a set of facts is NP-hard.*

It might seem rather depressing that forward chaining has an NP-hard matching problem in its inner loop. There are three ways to cheer ourselves up:

- We can remind ourselves that most rules in real-world knowledge bases are small and simple (like the rules in our crime example) rather than large and complex (like the CSP formulation in Figure 9.5). It is common in the database world to assume that both the sizes of rules and the arities of predicates are bounded by a constant and to worry only about **data complexity**—that is, the complexity of inference as a function of the number of ground facts in the database. It is easy to show that the data complexity of forward chaining is polynomial.
- We can consider subclasses of rules for which matching is efficient. Essentially every Datalog clause can be viewed as defining a CSP, so matching will be tractable just when the corresponding CSP is tractable. Chapter 5 describes several tractable families of CSPs. For example, if the constraint graph (the graph whose nodes are variables and whose links are constraints) forms a tree, then the CSP can be solved in linear time. Exactly the same result holds for rule matching. For instance, if we remove South Australia from the map in Figure 9.5, the resulting clause is

$$Diff(wa, nt) \wedge Diff(nt, q) \wedge Diff(q, nsw) \wedge Diff(nsw, v) \Rightarrow Colorable()$$

  which corresponds to the reduced CSP shown in Figure 5.11. Algorithms for solving tree-structured CSPs can be applied directly to the problem of rule matching.
- We can work hard to eliminate redundant rule matching attempts in the forward chaining algorithm, which is the subject of the next section.

Incremental forward chaining

When we showed how forward chaining works on the crime example, we cheated; in particular, we omitted some of the rule matching done by the algorithm shown in Figure 9.3. For example, on the second iteration, the rule

$$Missile(x) \Rightarrow Weapon(x)$$

matches against $Missile(M_1)$ (again), and of course the conclusion $Weapon(M_1)$ is already known so nothing happens. Such redundant rule matching can be avoided if we make the following observation: *Every new fact inferred on iteration t must be derived from at least one new fact inferred on iteration $t-1$.* This is true because any inference that does not require a new fact from iteration $t-1$ could have been done at iteration $t-1$ already.

This observation leads naturally to an incremental forward chaining algorithm where, at iteration t, we check a rule only if its premise includes a conjunct p_i that unifies with a fact p'_i newly inferred at iteration $t-1$. The rule matching step then fixes p_i to match with p'_i, but allows the other conjuncts of the rule to match with facts from any previous iteration. This algorithm generates exactly the same facts at each iteration as the algorithm in Figure 9.3, but is much more efficient.

With suitable indexing, it is easy to identify all the rules that can be triggered by any given fact, and indeed many real systems operate in an "update" mode wherein forward chaining occurs in response to each new fact that is TELLed to the system. Inferences cascade through the set of rules until the fixed point is reached, and then the process begins again for the next new fact.

Typically, only a small fraction of the rules in the knowledge base are actually triggered by the addition of a given fact. This means that a great deal of redundant work is done in constructing partial matches repeatedly that have some unsatisfied premises. Our crime example is rather too small to show this effectively, but notice that a partial match is constructed on the first iteration between the rule

$$American(x) \wedge Weapon(y) \wedge Sells(x, y, z) \wedge Hostile(z) \Rightarrow Criminal(x)$$

and the fact $American(West)$. This partial match is then discarded and rebuilt on the second iteration (when the rule succeeds). It would be better to retain and gradually complete the partial matches as new facts arrive, rather than discarding them.

The **rete** algorithm[3] was the first to address this problem seriously. The algorithm preprocesses the set of rules in the knowledge base to construct a sort of dataflow network in which each node is a literal from a rule premise. Variable bindings flow through the network and are filtered out when they fail to match a literal. If two literals in a rule share a variable—for example, $Sells(x, y, z) \wedge Hostile(z)$ in the crime example—then the bindings from each literal are filtered through an equality node. A variable binding reaching a node for an n-ary literal such as $Sells(x, y, z)$ might have to wait for bindings for the other variables to be established before the process can continue. At any given point, the state of a rete network captures all the partial matches of the rules, avoiding a great deal of recomputation.

Rete networks, and various improvements thereon, have been a key component of so-called **production systems**, which were among the earliest forward chaining systems in widespread use.[4] The XCON system (originally called R1, McDermott, 1982) was built using a production system architecture. XCON contained several thousand rules for designing configurations of computer components for customers of the Digital Equipment Corporation. It was one of the first clear commercial successes in the emerging field of expert systems. Many other similar systems have been built using the same underlying technology, which has been implemented in the general-purpose language OPS-5.

Production systems are also popular in **cognitive architectures**—that is, models of human reasoning—such as ACT (Anderson, 1983) and SOAR (Laird *et al.*, 1987). In such systems, the "working memory" of the system models human short-term memory, and the productions are part of long-term memory. On each cycle of operation, productions are matched against the working memory of facts. A production whose conditions are satisfied can add or delete facts in working memory. In contrast to the typical situation in databases, production systems often have many rules and relatively few facts. With suitably optimized matching technology, some modern systems can operate in real time with over a million rules.

Irrelevant facts

The final source of inefficiency in forward chaining appears to be intrinsic to the approach and also arises in the propositional context. (See Section 7.5.) Forward chaining makes all allowable inferences based on the known facts, *even if they are irrelevant to the goal at hand*. In our crime example, there were no rules capable of drawing irrelevant conclusions,

[3] Rete is Latin for net. The English pronunciation rhymes with treaty.

[4] The word **production** in **production systems** denotes a condition–action rule.

so the lack of directedness was not a problem. In other cases (e.g., if we have several rules describing the eating habits of Americans and the prices of missiles), FOL-FC-ASK will generate many irrelevant conclusions.

One way to avoid drawing irrelevant conclusions is to use backward chaining, as described in Section 9.4. Another solution is to restrict forward chaining to a selected subset of rules; this approach was discussed in the propositional context. A third approach has emerged in the deductive database community, where forward chaining is the standard tool. The idea is to rewrite the rule set, using information from the goal, so that only relevant variable bindings—those belonging to a so-called **magic set**—are considered during forward inference. For example, if the goal is $Criminal(West)$, the rule that concludes $Criminal(x)$ will be rewritten to include an extra conjunct that constrains the value of x:

$$Magic(x) \wedge American(x) \wedge Weapon(y) \wedge Sells(x, y, z) \wedge Hostile(z) \Rightarrow Criminal(x) .$$

The fact $Magic(West)$ is also added to the KB. In this way, even if the knowledge base contains facts about millions of Americans, only Colonel West will be considered during the forward inference process. The complete process for defining magic sets and rewriting the knowledge base is too complex to go into here, but the basic idea is to perform a sort of "generic" backward inference from the goal in order to work out which variable bindings need to be constrained. The magic sets approach can therefore be thought of as a kind of hybrid between forward inference and backward preprocessing.

BACKWARD CHAINING

The second major family of logical inference algorithms uses the **backward chaining** approach introduced in Section 7.5. These algorithms work backward from the goal, chaining through rules to find known facts that support the proof. We describe the basic algorithm, and then we describe how it is used in **logic programming**, which is the most widely used form of automated reasoning. We will also see that backward chaining has some disadvantages compared with forward chaining, and we look at ways to overcome them. Finally, we will look at the close connection between logic programming and constraint satisfaction problems.

A backward chaining algorithm

Figure 9.6 shows a simple backward-chaining algorithm, FOL-BC-ASK. It is called with a list of goals containing a single element, the original query, and returns the set of all substitutions satisfying the query. The list of goals can be thought of as a "stack" waiting to be worked on; if *all* of them can be satisfied, then the current branch of the proof succeeds. The algorithm takes the first goal in the list and finds every clause in the knowledge base whose positive literal, or **head**, unifies with the goal. Each such clause creates a new recursive call in which the premise, or **body**, of the clause is added to the goal stack. Remember that facts are clauses with a head but no body, so when a goal unifies with a known fact, no new subgoals are added to the stack and the goal is solved. Figure 9.7 is the proof tree for deriving $Criminal(West)$ from sentences (9.3) through (9.10).

```
function FOL-BC-ASK(KB, goals, θ) returns a set of substitutions
  inputs: KB, a knowledge base
          goals, a list of conjuncts forming a query (θ already applied)
          θ, the current substitution, initially the empty substitution { }
  local variables: answers, a set of substitutions, initially empty

  if goals is empty then return {θ}
  q' ← SUBST(θ, FIRST(goals))
  for each sentence r in KB where STANDARDIZE-APART(r) = (p1 ∧ ... ∧ pn ⇒ q)
        and θ' ← UNIFY(q, q') succeeds
    new_goals ← [p1, ..., pn | REST(goals)]
    answers ← FOL-BC-ASK(KB, new_goals, COMPOSE(θ', θ)) ∪ answers
  return answers
```

Figure 9.6 A simple backward-chaining algorithm.

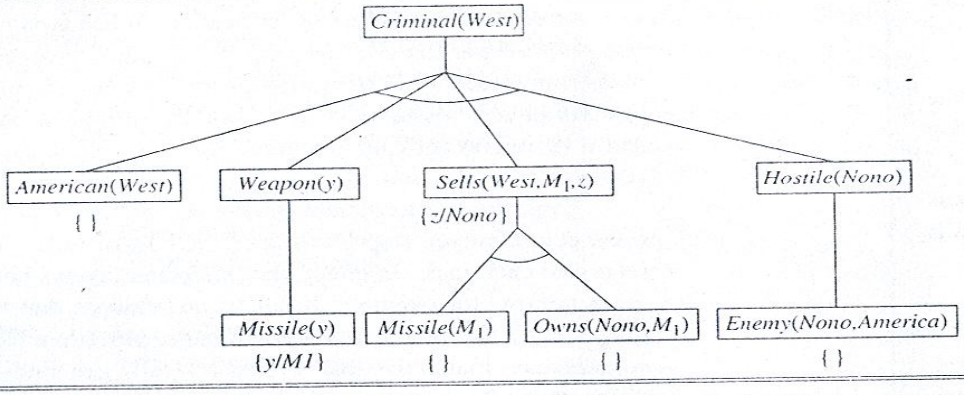


Figure 9.7 Proof tree constructed by backward chaining to prove that West is a criminal. The tree should be read depth first, left to right. To prove *Criminal*(*West*), we have to prove the four conjuncts below it. Some of these are in the knowledge base, and others require further backward chaining. Bindings for each successful unification are shown next to the corresponding subgoal. Note that once one subgoal in a conjunction succeeds, its substitution is applied to subsequent subgoals. Thus, by the time FOL-BC-ASK gets to the last conjunct, originally *Hostile*(z), z is already bound to *Nono*.

The algorithm uses **composition** of substitutions. COMPOSE(θ_1, θ_2) is the substitution whose effect is identical to the effect of applying each substitution in turn. That is,

$$\text{SUBST}(\text{COMPOSE}(\theta_1, \theta_2), p) = \text{SUBST}(\theta_2, \text{SUBST}(\theta_1, p))$$

In the algorithm, the current variable bindings, which are stored in θ, are composed with the bindings resulting from unifying the goal with the clause head, giving a new set of current bindings for the recursive call.

Backward chaining, as we have written it, is clearly a depth-first search algorithm. This means that its space requirements are linear in the size of the proof (neglecting, for now, the space required to accumulate the solutions). It also means that backward chaining (unlike forward chaining) suffers from problems with repeated states and incompleteness. We will discuss these problems and some potential solutions, but first we will see how backward chaining is used in logic programming systems.

Logic programming

Logic programming is a technology that comes fairly close to embodying the declarative ideal described in Chapter 7: that systems should be constructed by expressing knowledge in a formal language and that problems should be solved by running inference processes on that knowledge. The ideal is summed up in Robert Kowalski's equation,

$$Algorithm = Logic + Control .$$

Prolog is by far the most widely used logic programming language. Its users number in the hundreds of thousands. It is used primarily as a rapid-prototyping language and for symbol-manipulation tasks such as writing compilers (Van Roy, 1990) and parsing natural language (Pereira and Warren, 1980). Many expert systems have been written in Prolog for legal, medical, financial, and other domains.

Prolog programs are sets of definite clauses written in a notation somewhat different from standard first-order logic. Prolog uses uppercase letters for variables and lowercase for constants. Clauses are written with the head preceding the body; ":-" is used for left-implication, commas separate literals in the body, and a period marks the end of a sentence:

```
criminal(X) :- american(X), weapon(Y), sells(X,Y,Z), hostile(Z).
```

Prolog includes "syntactic sugar" for list notation and arithmetic. As an example, here is a Prolog program for `append(X,Y,Z)`, which succeeds if list `Z` is the result of appending lists `X` and `Y`:

```
append([],Y,Y).
append([A|X],Y,[A|Z]) :- append(X,Y,Z).
```

In English, we can read these clauses as (1) appending an empty list with a list `Y` produces the same list `Y` and (2) `[A|Z]` is the result of appending `[A|X]` onto `Y`, provided that `Z` is the result of appending `X` onto `Y`. This definition of `append` appears fairly similar to the corresponding definition in Lisp, but is actually much more powerful. For example, we can ask the query `append(A,B,[1,2])`: what two lists can be appended to give `[1,2]`? We get back the solutions

```
A=[]    B=[1,2]
A=[1]   B=[2]
A=[1,2] B=[]
```

The execution of Prolog programs is done via depth-first backward chaining, where clauses are tried in the order in which they are written in the knowledge base. Some aspects of Prolog fall outside standard logical inference:

- There is a set of built-in functions for arithmetic. Literals using these function symbols are "proved" by executing code rather than doing further inference. For example, the goal "`X is 4+3`" succeeds with `X` bound to 7. On the other hand, the goal "`5 is X+Y`" fails, because the built-in functions do not do arbitrary equation solving.[5]
- There are built-in predicates that have side effects when executed. These include input–output predicates and the `assert/retract` predicates for modifying the knowledge base. Such predicates have no counterpart in logic and can produce some confusing effects—for example, if facts are asserted in a branch of the proof tree that eventually fails.
- Prolog allows a form of negation called **negation as failure**. A negated goal `not P` is considered proved if the system fails to prove `P`. Thus, the sentence

  ```
  alive(X) :- not dead(X).
  ```

  can be read as "Everyone is alive if not provably dead."
- Prolog has an equality operator, `=`, but it lacks the full power of logical equality. An equality goal succeeds if the two terms are *unifiable* and fails otherwise. So `X+Y=2+3` succeeds with `X` bound to `2` and `Y` bound to `3`, but `morningstar=eveningstar` fails. (In classical logic, the latter equality might or might not be true.) No facts or rules about equality can be asserted.
- The **occur check** is omitted from Prolog's unification algorithm. This means that some unsound inferences can be made; these are seldom a problem except when using Prolog for mathematical theorem proving.

The decisions made in the design of Prolog represent a compromise between declarativeness and execution efficiency—inasmuch as efficiency was understood at the time Prolog was designed. We will return to this subject after looking at how Prolog is implemented.

Efficient implementation of logic programs

The execution of a Prolog program can happen in two modes: interpreted and compiled. Interpretation essentially amounts to running the FOL-BC-ASK algorithm from Figure 9.6, with the program as the knowledge base. We say "essentially," because Prolog interpreters contain a variety of improvements designed to maximize speed. Here we consider only two.

First, instead of constructing the list of all possible answers for each subgoal before continuing to the next, Prolog interpreters generate one answer and a "promise" to generate the rest when the current answer has been fully explored. This promise is called a **choice point**. When the depth-first search completes its exploration of the possible solutions arising from the current answer and backs up to the choice point, the choice point is expanded to yield a new answer for the subgoal and a new choice point. This approach saves both time and space. It also provides a very simple interface for debugging because at all times there is only a single solution path under consideration.

Second, our simple implementation of FOL-BC-A