Many problems that arose in earlier chapters were not resolved because they required knowledge of context. Two important aspects of context are general knowledge about the world and specific knowledge about the situation in which the linguistic communication is occurring. To analyze these, you need a formalism for representing knowledge and reasoning. This area of study is called **knowledge representation (KR)**.

Knowledge representation means different things to different researchers. For some, knowledge representation concerns the structure of the language used to express the knowledge—whether it is in logic, semantic networks, frames, or other specially designed representational formalisms. For others, knowledge representation concerns the content of sentences—what predicates are needed and how are they organized. Both of these issues are important. Sometimes, what seems to be a vigorous debate about knowledge representation is actually the result of each of the debaters focusing on one of the aspects of representation without considering the concerns of the other.

There is not the space here for an extended discussion of knowledge representation formalisms. Rather, the central concern is how knowledge and reasoning can be used to facilitate language understanding. Because of this, an abstracted representation based on the first-order predicate calculus will be used. This will enable the discussion of relevant representational issues with the minimum introduction of new material and notation. This does not mean that the underlying knowledge representation used in a system would have to directly represent logical formulas or use theorem-proving techniques as a model of inference. The underlying representation system could be a semantic network, a description logic, a frame-based system, a connectionist model, or any other formalism, as long as the system has at least the expressive power described here. Many modern knowledge representation systems fulfill this requirement.

Section 13.1 discusses some general issues in knowledge representation. The next two sections develop the abstract knowledge representation language that will be used throughout the rest of the book: Section 13.2 discusses a simple representation based on the FOPC, and Section 13.3 discusses a framelike representation as a way of clustering information and representing defaults about stereotypical objects and situations. Section 13.4 discusses an important issue in mapping the logical form language to the knowledge representation language, namely the representation of the complex quantifiers found in natural language. The rest of the chapter is optional but is important if your focus is knowledge representation. Section 13.5 examines the representation of temporal information in language, as revealed by tense and aspectual class. The remaining sections give examples of some different reasoning strategies that are used in knowledge representation systems. Section 13.6 discusses some basic concepts in automated deduction, which are used in systems based on deductive techniques as well as systems based on logic programming. Section 13.7 discusses an approach that uses a procedural semantics and illustrates its use in a question-answering system. Section 13.8 discusses some issues in developing hybrid reasoning systems, which allow different reasoning techniques to be used depending on what sort of information is required.

This chapter assumes that the reader has a basic understanding of the first-order predicate calculus, at least to the level introduced in Appendix A.

13.1 Knowledge Representation

There are two forms of knowledge that are crucial in any knowledge representation system: general knowledge of the world and specific knowledge of the current situation. Some aspects of general world knowledge have already been considered, such as type hierarchies, part/whole relationships, and so on. This consists of information about general constraints on the world and the semantic definition of the terms in the language. For the most part, general knowledge is specified in terms of the **types** or kinds of objects in the world and does not concern information about specific individuals. For instance, it might encode that OWN1 is a relation between people and objects but not that a particular person, say John, owns a particular car. This latter information, knowledge about individuals, is equally important in language understanding and is a major component of what we call the specific setting of the sentence being understood. Virtually all knowledge representation systems support reasoning at both of these levels in one way or another.

General world knowledge is essential for solving many language interpretation problems, one of the most important being disambiguation. For example, the proper attachment of the final PP in the following two sentences depends solely on the reader's background knowledge of the appropriate time needed for reading and for evolution:

I read a story about evolution in ten minutes.
I read a story about evolution in the last million years.

Specific knowledge of the situation is important for many issues, including determining the referent of noun phrases and disambiguating word senses based on what makes sense in the current situation.

We will sometimes talk informally of the knowledge representation as encoding the knowledge and beliefs of the understanding system. But since the terms knowledge and belief will be given more precise technical definitions later, more neutral terminology is introduced. A knowledge representation consists of a database of sentences called the **knowledge base** (KB) and a set of **inference techniques** that can be used to derive new sentences given the current KB. A set of inference techniques is **sound** if it only derives true new sentences when the original sentences in the KB are all true. Not all useful inference techniques need be sound, however, as you will see later.

The language in which the sentences in the KB are defined is called the **knowledge representation language** (KRL). The KRL could be the same as the logical form language, but there are practical reasons why they often differ. The

two languages are driven by different needs. The logical form language must be very expressive in order to simplify the semantic interpretation process and to allow the effective resolution of ambiguity. The knowledge representation language, on the other hand, must support efficient and predictable reasoning within a specific domain. In other words, it should be relatively easy to define the set of sentences that can be inferred from a particular KB and to build computational models that perform such inferences in a reasonable amount of time.

For example, consider the treatment of quantifiers. In the logical form language a wide range of quantifiers was introduced, closely corresponding to the different word senses of English quantifiers. This allows disambiguation techniques, such as those needed for scoping quantifiers, to use subtle differences between the actual quantifiers (say between *each* and *every*). In most current knowledge representation languages, however, there are usually only a few quantifiers and often only one—a construct allowing universal quantification. Thus inference processes can be easily defined. By keeping the languages separate and defining a mapping function between them, you can have the advantages of both. Of course, the success of this approach will depend on whether or not the mapping function can be effectively defined.

Substantial research will be needed before the best way to satisfy the need for an expressive logical form and an effective knowledge representation can be determined. Given the current state of knowledge, maintaining a separate logical form and knowledge representation languages seems the best compromise.

When a formula P must be true given the formulas in a KB, or given formulas representing the meaning of a sentence, then we say that the KB (or sentence) **entails** P. Many of the conclusions that need to be drawn to understand language are not entailments, however, but are **implications** of the sentence. Implications are conclusions that can typically be drawn from a sentence but that could be explicitly denied in specific circumstances. For instance, the sentence *Jack owns two cars* entails that Jack owns a car (that is, this fact cannot be denied), but only implies that he doesn't own three cars, as you could continue by saying *In fact, he owns three cars.* KR systems must support both these forms of inference.

Types of Inference

Many different forms of inference are necessary to understand natural language. Inference techniques can be classified into **deductive** and **nondeductive** forms. Deductive forms of inference are justified by the logical notion of entailment. Given a set of facts, a deductive inference process will make only conclusions that logically follow from those facts. Nondeductive inference falls into several classes. Examples include inference techniques that involve learning generalities from examples (**inductive inference**) and techniques that involve inferring causes from effects (a form of **abductive inference**).

Abductive inference can be contrasted with deductive inference by considering the axiom

$A \supset B$

Deductive inference would use this axiom to infer B when given A. Abductive inference would use it to infer A when given B, since A is a reason that B is true.

Many systems allow the use of default information. A **default rule** is an inference rule to which there may be exceptions; thus it is **defeasible**. If you write default information using the notation $A \Rightarrow B$, then the default inference rule could be stated as follows: If $A \Rightarrow B$, and A is true, and $\neg B$ is not provable, then conclude B. It has been suggested that default rules may provide a good account of generic sentences. For example, the meaning of the sentence *Birds fly* could be represented by the FOPC formula

$\forall\, x\, BIRD(x) \Rightarrow FLIES(x)$

This has the effect that whenever there is a bird *B* for which it is not provable that $\neg FLIES(B)$, then it can be inferred that $FLIES(B)$. In other words, a specific bird will be assumed to be able to fly unless it is explicitly stated that it cannot.

Defeasible rules introduce a new set of complexities in a representation. Without such rules, most representations are **monotonic**, because adding new assertions only increases the number of formulas entailed. Specifically, in a monotonic representation, if the knowledge base KB1 entails a conclusion C, and if you add an additional formula to KB1 to form a new consistent knowledge base KB2, then KB2 will also entail C. This is not true of a representation that uses default rules, and hence they are called **nonmonotonic** representations. For example, consider a knowledge base K consisting of the formulas

$Cat(Sampson)$	Sampson is a cat.
$TabbyCat(Sampson)$	Sampson is a tabby cat.
$\forall\, c\, .\, Cat(c) \Rightarrow Purrs(c)$	Cats purr.

Given this KB, you can conclude *Purrs*(*Sampson*) using the default rule because there is no information to contradict *Purrs*(*S*). On the other hand, if you add a new fact that no tabby cats purr, then the extended knowledge base would no longer entail that Sampson purrs.

There are other useful techniques for introducing nonmonotonic conclusions besides default rules. For instance, the **closed world assumption (CWA)** asserts that the KB contains complete information about certain predicates. For example, for a predicate P for which the CWA holds, if a proposition involving P cannot be proven from a KB, then its negation is assumed to be true. Consider a database query application for airline schedules. The KB stores information about flights that exist—say, that flight FDG100 flies from Rochester to Boston—but it doesn't explicitly contain negative information—say, that flight FDG100 doesn't fly to Chicago or that there is no flight FDG455. Such information can only be concluded if the inference process makes the closed world assumption on flights.

BOX 13.1 A Semantics for Nonmonotonic Logic

You can develop a model theoretic semantics for many nonmonotonic constructs using the concept of **minimal models**. For example, consider the closed world assumption for a predicate P. We can define an ordering on all the models of the KB as follows:

$\mathbf{m1} <_P \mathbf{m2}$ iff $I_{m1}(P) \subseteq I_{m2}(P)$

In other words, a model **m1** is smaller than a model **m2** with respect to a predicate P if and only if the set of objects x such that P(x) is true in **m1** is a subset of the set of objects x such that P(x) is true in **m2**. With this ordering defined, the minimal models with respect to P consist of the set of models $\{m \mid \text{there is no } m' <_P m\}$. Given a suitable knowledge base K, it can be shown that the conclusions derivable from K making the closed world assumption on P are exactly the conclusions that are entailed by the minimal models (with respect to P) of K.

Another way to formalize the closed world assumption is to add an axiom to the KB that specifically entails the closure. This axiom is called the **predicate completion axiom**. Consider a KB containing the propositions

$P(A), P(B), Q(C), Q(A)$

The predicate completion axiom for P would be

$\forall\, x\, .\, P(x) \equiv (x{=}A \vee x{=}B)$

This axiom would allow you to conclude that P is only true for A and B. Thus you could infer $\neg P(C)$, as desired. It can be shown that the set of models for the KB extended with the predicate completion axioms is exactly the set of minimal models (with respect to P) of the initial KB. Predicate completion axioms can handle more complex KBs as well. For instance, if the KB also included the axiom

$\forall\, s\, .\, Q(s) \supset P(s),$

then the predicate completion axiom for P would be

$\forall\, x\, .\, P(x) \equiv (x{=}A \vee x{=}B \vee Q(x))$

Predicate completion cannot be applied to all KBs, however, as it can't handle axioms that contain more than one positive occurrence of P. A generalization, called **circumscription** (McCarthy, 1980), can generate an appropriate closure axiom, but may require using a second-order logic (that is, involving quantification over predicates).

Inference Techniques

The two main classes of inference techniques found in knowledge representation systems are **procedural** and **declarative**. Most systems combine these techniques to some extent, forming a continuum from purely declarative

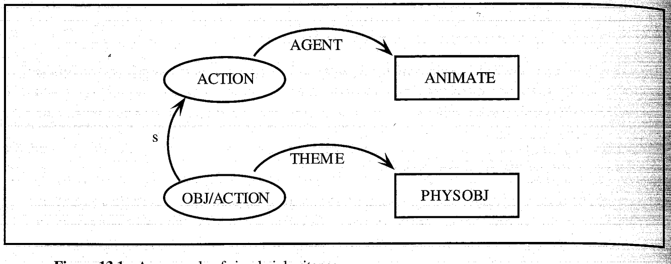

Figure 13.1 An example of simple inheritance

representations to purely procedural ones. The declarative end of the continuum would be a logic-based theorem prover. The KB is represented as a set of axioms, and inference is performed using a deductive theorem-proving algorithm. In a strongly declarative system, the emphasis is on assigning a formal semantics to the expressions of the representation independent of the inference component.

Procedural inference systems, on the other hand, emphasize the inferential aspects of the representation, and in extreme cases the expressions in the KB may be given no meaning independent of how they are manipulated by the program. An example of a procedural representation might be a system that uses the computer's own built-in arithmetic procedures to evaluate arithmetic expressions without any explicit representation of knowledge about mathematics, such as Peano's axioms of arithmetic. In practice, procedural systems can be very effective at specific inference tasks in well-defined domains but are often hard to analyze because they lack formality.

Consider an example. In Chapter 10 the technique of inheritance was introduced for semantic networks. This inference process can be realized procedurally or declaratively. A purely declarative approach would model each fact about subtypes and roles as an axiom and the inheritance properties would result from standard deductive inference. For example, given the simple network in Figure 13.1, the following FOPC axioms might represent this information:

1. $\forall$ x . ACTION(x) $\supset$ $\exists$ a . AGENT(x, a) & ANIMATE(a)
2. $\forall$ a $\exists$ x . ACTION(x) & AGENT(x, a) $\supset$ ANIMATE(a)
3. $\forall$ x . OBJ/ACTION(x) $\supset$ ACTION(x)
4. $\forall$ x . OBJ/ACTION(x) $\supset$ $\exists$ o . THEME(x, o) & PHYSOBJ(o)
5. $\forall$ o $\exists$ x . OBJ/ACTION(x) & THEME(x, o) $\supset$ PHYSOBJ(o)

Using these axioms, you can prove that the class OBJ/ACTION "inherits" the AGENT role. In other words, for any object A such that *OBJ/ACTION(A)* is true, you could prove that A has an agent role; that is,

$\exists$ *a* . *AGENT(A, a)* & *ANIMATE(a)*

using axioms 3 and 1.

As described in Chapter 10, a procedural version of this would be a program that starts at the specified node OBJ/ACTION, finds all roles attached at that node, and then follows the S arc up to the supertype ACTION and finds all the roles attached there. The complete set of roles gathered by this procedure is the answer. Thus any OBJ/ACTION has an AGENT role inherited from the class ACTION.

Both these techniques compute the same result, but the first does it by using deduction over logical formulas, while the second uses a program that performs a graph traversal. The first technique seems more rigorously defined, but the second is probably more efficient. In cases like this, in which you can prove that the two techniques obtain the same results, you can have the best of both approaches: a rigorously defined semantics and an efficient procedure to perform that form of inference.

13.2 A Representation Based on FOPC

The KRL used in this book will be an extended version of the first-order predicate calculus. Note that by choosing the language, you are not committed to any particular form of inference. For example, later sections will show how the KRL can be used with both deductive and procedural inference techniques.

The syntax of FOPC was introduced earlier and will not be presented again here. We will focus on the extensions to standard FOPC that are needed to represent the meaning of natural language sentences, and comment on the differences between this language and the logical form language. The terms of the language consist of constants (such as *John1*), functions (such as *father*(*John1*)), and variables (such as *x* and *y*). Note that the logical form language did not use constants. Rather, everything was expressed in terms of discourse variables to keep the representation context independent. In the KB, constants are used to represent the specific individuals. For example, the logical form term (NAME **j1** "John") represents the meaning of a phrase whose referent is named "John." The actual person referred to in a given context might be represented by the constant *John1* in the KB.

It is convenient to use restricted quantification in the KRL, making it similar to the generalized quantifier notation in the logical form language. Restrictions follow the quantified variable separated by a colon. As mentioned in Chapter 8, for the existential and universal quantifiers, this notation can be treated as an abbreviation and does not extend the expressive power of the language. Thus

$\exists\ x : Man(x)\ Happy(x)$ is equivalent to $\exists$ x . $Man(x)\ \&\ Happy(x)$ and
$\forall\ x : Man(x)\ Happy(x)$ is equivalent to $\forall\ x . Man(x) \supset Happy(x)$

We will also need the *equality* predicate, ($a = b$), which states that terms *a* and *b* have the same denotation. Given a simple proposition P_a involving a constant *a*, if P_a is true and $a = b$, then P_b must be true as well, where P_b is the same as P_a except that a has been replaced by b.

Many knowledge representation systems do not explicitly use quantifiers. They do include variables, however, which act like universally quantified variables with wide scope. For example, a formula such as (P ?x A) in a KB would correspond in meaning to the FOPC formula $\forall\ x\ .\ P(x, A)$. Existentially quantified variables are handled by a technique called **skolemization**, which replaces the variable with a new constant that has not been used before. For example, the formula $\exists\ y\ \forall\ x\ .\ P(x, y)$ would be encoded in the KB in a formula such as (P ?x Sk1), where Sk1 is a new constant that has not been used before, that stands for the object that is known to exist. Quantifier scoping dependencies are indicated using new functions, called **Skolem functions**. For example, the formula $\forall\ y\ \exists\ x\ .\ P(x, y)$ would be encoded as a formula such as (P (Sk2 ?y) ?y) in the KB, where Sk2 is a new function that produces a (potentially) different object for each value of ?y. Often, formulas will be written in a format merging these two approaches, where the universal quantifiers are still present but the existential variables have been skolemized. For example, the formula $\forall\ y\ \exists\ x\ .\ P(x, y)$ may be written as $\forall\ y\ P(Sk1(y), y)$. It can be proven that all these different forms of representation are equivalent.

Saying that the basic representation language is FOPC does not place many restrictions on the style of the representation. In particular, it says nothing about what the predicates are. There is a wide range of possibilities in selecting the predicates. At one end you could have a different predicate for each word sense, essentially the strategy used in the logical form language. At the other end you could have a preset set of predicates, called the **primitives**, and every word sense would have to be defined in terms of these primitives. Consider some of the advantages of each position. By allowing a predicate for each word sense, you are able to capture subtle differences between semantically close terms. For example, you might have information in the KB defining SAUNTERS1 as an action that involves walking slowly, using a manner that suggests a carefree state of mind. Thus the sentence *Jack sauntered down the street* might have different implications than *Jack walked down the street.* Of course, you pay for this power by having a wide range of predicates that for the most part have very similar axioms defining them. Specifically, the definitions for SAUNTERS1 and WALKS1 would overlap significantly.

In an approach using primitives, on the other hand, both of these senses would be reduced to the predicate (or set of predicates) that captures the basic action, say MOVE-BY-FOOT. Inference rules are then defined only on the primitive predicates. This approach allows the commonalities between words to be captured very succinctly. Without inference rules on the word senses, however, it is very difficult to capture the subtle distinctions between senses. Of course, you would have to define a new primitive to capture the distinction between sauntering and walking, say a new primitive concerning state-of-mind. Sauntering might then be defined as MOVE-BY-FOOT and CAREFREE-STATE. The more

complex the decomposition of the senses, however, the less advantageous the primitive representation becomes, because the number of primitives grows significantly, driven by examples. More crucially, inference rules would have to be based on complex clusters of primitives rather than single predicates, so the inference process is no longer so simply defined.

As with many issues, there is considerable middle ground to be explored. Specifically, many of the advantages of a primitive-based representation can be captured by using type hierarchies. If you assert that SAUNTERS1 and WALKS1 are both subclasses of the more abstract action MOVE-BY-FOOT, then they could inherit most of their common properties from MOVE-BY-FOOTwithout the need for additional axioms. This still leaves you free, however, to add other axioms for SAUNTERS1 to cover its special characteristics.

This approach would also allow you to handle incomplete knowledge. Say the system only knows that SAUNTERS1 is a type of walking. It doesn't know any additional information about the word but can still make most inferences required about it using information inherited from MOVE-BY-FOOT. In addition, it knows that SAUNTERS1 is somehow different from WALKS1, even though if doesn't know why. If, at a later stage, the system acquires additional knowledge about sauntering, this can be added incrementally.

In addition to hierarchical relations, a knowledge representation should also be able to take advantage of other ways to define word senses. Sometimes a complete definition of a term is known. For example, you could define the predicate *father* as a *male parent,* that is,

$$\forall\ x\ .\ FATHER(x) \equiv \exists\ y\ PARENT(x, y)\ \&\ MALE(x)$$

But most words are not so precisely defined. For instance, there is no set of properties that precisely defines most natural kinds, such as dogs, cats, chairs, and so on. These can be classified into type hierarchies, and axioms stating necessary conditions can be stated, but no absolute definition is possible. Viewed as FOPC axioms, this means that such definitions involve a one-way implication. For instance, an axiom for DOG1 might be

$$\forall\ x\ .\ DOG1(x) \supset CANINE(x)\ \&\ DOMESTIC\text{-}PET(x)$$

where CANINE itself is defined as a type of MAMMAL, and so on. Such axioms capture much of the important properties of being a dog but do not define the concept completely. For instance, someone might have a pet wolf that satisfies all the properties of being a dog but still isn't a dog.

From the point of view of generating sentences, the more the predicates in representation language are abstracted from the words in the language, the harder it is to produce sentences based on meanings. For instance, assume you are given the formula

$$\forall\ p : ((MaleHuman\ p)\ \&\ \exists\ c\ .\ Parent(p, c))\ .$$
$$MoveByCar(p, L1)\ \&\ Building(L1)\ \&\ Used\text{-}for\text{-}teaching(L1)$$

which has a natural realization as the sentence *All fathers drove to the school.* To generate such a sentence, the system would have to be able to realize the formula ((*Male p*) & ∃ *c* . *Parent*(*p, c*)), which literally might be realized as *male humans who have a child,* as the word *father,* and realize the proposition *MoveByCar*(*p, L1*) & *Building*(*L1*) & *Used-for-teaching*(*L1*), which literally might be realized as *moved by car to a building used for teaching,* as the phrase *drove to school.* Clearly, this would require substantial knowledge about the meanings of the specific words *father* and *drive,* and a complex process of matching formulas in the KRL to these predicates. If there are no predicates corresponding to these word meanings in the KRL, then this process is especially complicated. If such predicates are included, the hierarchical organization would suggest methods for identifying possible realizations of a formula. Specifically, given an abstract predicate, say *MaleHuman,* you could consider all the predicates below it in the abstraction hierarchy to see if any of them more concisely capture the desired meaning. In this case *Father* would be a good choice as it not only entails *MaleHuman* but also another part of the meaning, namely ∃ *c* . *Parent*(*p, c*).

The trick in designing an effective knowledge representation is to choose the set of predicates so as to make the hierarchical relationships most effective. Often, the best representation will mirror linguistic generalizations that can be made. This aids both in interpreting sentences and in generating sentences from expressions in the KB.

13.3 Frames: Representing Stereotypical Information

Much of the inference required for natural language understanding involves making assumptions about what is typically true of the objects or situations being discussed. Such information is often encoded in structures called **frames**. In its most abstract formulation, a frame is simply a cluster of facts and objects that describe some typical object or situation, together with specific inference strategies for reasoning about the situation. The situations represented could range from visual scenes, to the structure of complex physical objects, to the typical method by which some action is performed. Frame-based systems usually offer facilities such as default reasoning, automatic inheritance of properties through hierarchies, and procedural attachment. In some implementations all reasoning is accomplished by specialized inference procedures attached to the frame; in others the frames are mostly declarative in nature and are interpreted by a more uniform inference procedure. Either way, the key idea is the clustering of information to characterize the properties of commonly occurring objects and situations.

The principal objects in a frame are assigned names, called **slots** or **roles** (similar to the thematic roles in the logical form). For instance, the frame for a house may have slots such as kitchen, living room, hallway, front door, and so on. The frame also specifies the relationships between the slots and the object represented by the frame. For example, the kitchen slot of the house frame has to be physically located within the house, and it contains various appliances needed

BOX 13.2 Conceptual Dependency: A Primitive-Based Representation

Several very influential early semantic representations were based on small sets of primitives that were used to support a set of specialized reasoning techniques. One of the most influential was **conceptual dependency** (Schank, 1975; Schank and Riesbeck, 1981). This representation primarily focused on action verbs and posited a small set of action types. Specifically, the major action types included three notions of transfer:

ATRANS—abstract transfer (as in transfer of ownership)
PTRANS—physical transfer
MTRANS—mental transfer (as in speaking)

There were also primitives based on bodily activity,

PROPEL (applying force)
MOVE (moving a body part)
GRASP
INGEST
EXPEL

as well as the mental actions,

CONC (conceptualize or think)
MBUILD (perform inference)

These primitives, together with a set of case roles and a few causal connectives, essentially completed the representation. In early works, it was claimed that this representation was adequate to express the meaning of all action verbs, but in later work primitives were used as building blocks to construct larger structures to capture the meaning of verbs (for example, see Section 15.5). It was found that inference had to be specified in terms of these larger structures rather than in terms of the primitives. Thus the advantages of the primitive-based representation were lost.

for preparing meals. You can view each of these slots as a function that takes an object described by the frame (an **instance** of the frame) and produces the appropriate slot value. Thus a particular instance of the house frame—say, H1—consists of a particular instance of a kitchen, which can be referred to as "the kitchen-slot of H1," or *kitchen*(*H1*), plus particular instances of all the other slots as well.

As an example, the definition of a frame type for personal computers might look as follows:

Define Object Class ***PC*(*e*)**:
Roles: *Keyb, Disk1, MainBox*
Constraints: *Keyboard*(*Keyb*), *DiskDrive*(*Disk1*), *CPU*(*MainBox*)

This structure means that all objects of type PC have slots of type keyboard, disk drive, and CPU (which are identified by the functions *Keyb, Disk1,* and *MainBox,*

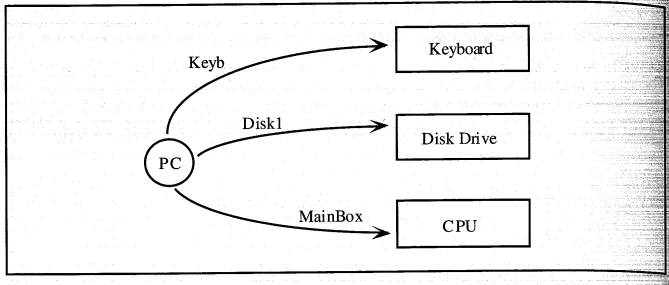

Figure 13.2 A semantic network defining the slots of PC

respectively). This is the same style of representation used in many semantic network systems. In fact, you could easily represent this structure in a semantic network notation as well, as shown in Figure 13.2.

An instance of the type *PC*—say, *PC3*—having the subparts *KEYS13, DD11*, and *CPU00023* would be represented in the frame notation as

(*PC3* isa *PC* with *Keyb* = *KEY13*, *Diskl* = *DD11*, *MainBox* = *CPU00023*)

This definition can be viewed as an abbreviation of the FOPC formula *PC*(*PC3*) & *Keyb*(*PC3*) = *KEY13* & *Diskl*(*PC3*) = *DD11* & *MainBox*(*PC3*) = *CPU00023*.

Slots with Restrictions

In general, you need more than a superficial knowledge of the structural components of a PC. For instance, the PC frame might contain more information about how the slot values typically interrelate. You might want to assert that each slot is a subpart and indicate how the parts are connected: the keyboard, for example, as well as the disk drive, plug into the CPU box at the appropriate connector. To assert this, you would have to define the CPU itself as a frame structure with slots such as *KeyboardPlug, DiskPort, PowerPlug,* and so on. The notation is extended as in the following example that redefines the class of PCs so that the keyboard and disk are subparts and are connected to the CPU.

Define Object Class ***PC*(*p*)**:
Roles: *Keyb, Diskl, MainBox*
Constraints: *Keyboard*(*Keyb*) & *PART-OF*(*Keyb, p*) &
CONNECTED-TO(*Keyb, KeyboardPlug*(*MainBox*)) &
DiskDrive(*Diskl*) & *PART-OF*(*Diskl, p*) &
CONNECTED-TO(*Diskl*, *DiskPort*(*MainBox*)) &
CPU(*MainBox*) & *PART-OF*(*MainBox*, *p*)

With this definition, an instance of PC—say, *PC4*—with slot values *KEY14, DD12,* and *CPU07,* would be written as

(*PC4* isa *PC* with *Keyb* = *KEY14*, *Diskl* = *DD12*, *MainBox* = *CPU07*)

which implies all the following information:

PC(*PC4*) & *Keyb*(*PC4*) = *KEY14* & *PART-OF*(*KEY14, PC4*) &
CONNECTED-TO(*KEY14*, *KeyboardPlug*(*CPU07*)) &
Diskl(*PC4*) = *DD12* & *PART-OF*(*DD12, PC4*) &
CONNECTED-TO(*DD12*, *DiskPort*(*CPU07*)) &
MainBox(*PC4*) = *CPU07* & *PART-OF*(*CPU07*, *PC4*)

Since frames are a way of encoding knowledge about classes of objects, it makes sense that frame information should be inherited through the type hierarchy. For example, if you define a subtype of *PCs* called *PC-With-Second-Disk,* this type should inherit all the slots of *PC.* If you define a new slot for this type—say, *Disk2*—then all instances will have four slots: *Keyb, Diskl, MainBox,* and *Disk2.*

Note that the information in a frame should be viewed as default conditions. For instance, it is possible to have a *PC,* say *PC5,* in which the keyboard is not connected to the computer. The fact that this property is violated does not make *PC5* fall out of the class of *PCs*; it just isn't a typical *PC.*

Frame-based representation can be used to encode additional information about situations beyond their subcomponents. One of the most useful examples of this for natural language understanding occurs in representing actions. As you will see in later chapters, knowledge about the usual situations in which actions occur can be very useful in interpreting language. In particular, knowledge about causality—what effects an action typically has and what conditions are typically necessary for the action to occur—are very important. The slot notation is extended to allow relations between the instance of the frame and other propositions or events. For actions, the following relations are useful:

preconditions—properties that typically enable the action,
effects—properties that are typically caused by the action,
decomposition—the way in which an action is typically performed (usually defined in terms of a sequence of subactions).

For example, Figure 13.3 shows the definition of the action of buying something. The action involves four objects: the buyer, the seller, the object, and an amount of money equal to the price of the object. Furthermore, the definition states that a purchase action can occur only when the buyer has enough money and the seller has the object (the preconditions), and that typically at the end the buyer owns the object and the seller has the money (the effects). Finally, a typical way something is purchased involves the buyer giving the seller the money and the seller giving the buyer the object (the decomposition). While this might seem to be quite mundane everyday information, such knowledge is crucial for understanding the

The Action Class ***BUY*(*b*)**:
Roles: *Buyer, Seller, Object, Money*
Constraints: *Human*(*Buyer*), *SalesAgent*(*Seller*), *IsObject*(*Object*)
Value(*Money*, *Price*(*Object*))
Preconditions: *OWNS*(*Buyer*, *Money*)
OWNS(*Seller, Object*)
Effects: ¬*OWNS*(*Buyer, Money*)
¬*OWNS*(*Seller, Object*)
OWNS(*Buyer, Object*)
OWNS(*Seller, Money*)
Decomposition: *GIVE*(*Buyer, Seller, Money*)
GIVE(*Seller, Buyer, Object*)

Figure 13.3 The definition of BUY with its decomposition

connections between actions and states described in sentences, which in turn are crucial for ambiguity resolution.

13. 4 Handling Natural Language Quantification

With the basic KRL defined, you can now consider some issues in mapping the logical form language into the KRL. One of the most obvious differences between the two languages is the treatment of quantifiers. The logical form contains a wide range of quantificational forms corresponding to the English quantifiers, while the KRL allows only universal and existential quantification. Reconciling this difference seems almost hopeless at first glance. Significant progress can be made to reduce the differences, however, by extending the ontology of the KRL to allow sets as objects.

A set is a collection of objects viewed as a unit. While sets in general may be finite (such as the set consisting of John and Mary) or infinite (such as the set of numbers greater than 7), we will only use finite sets in the KRL. A set can be indicated by listing its members in curly brackets; for example, $\{John1\ Mary1\}$ refers to the set consisting of the denotation of *John1* and the denotation of *Mary1*. The order doesn't matter; $\{John1\ Mary1\} = \{Mary1\ John1\}$. We also allow constants to denote sets. Thus *S1* might be a set defined by the formula $S1 = \{John1\ Mary1\}$. Full set theory would allow sets to be members of other sets. We will not use such sets in the KRL. Sets will usually be defined in terms of some property. This will be written in the form $\{y \mid P_y\}$, which is the set of all objects that satisfy the expression P_y. The set of all men is $\{y \mid Man(y)\}$. In addition, we introduce the following predicates to relate sets and individuals:

$S1 \subset S2$ iff all the elements of *S1* are in *S2*
$x \in S$ iff x is a member of the set S

With setlike objects in the representation, we can produce an interpretation for *Some men met at three,* as follows:

$$\exists M: M \subset \{x \mid Man(x)\}\ .\ Meet1(M, 3PM)$$

that is, there is a subset of men M that met at three. By convention, we will always use uppercase names for variables ranging over sets. In principle, sets are allowed in all situations where individuals have been allowed. In practice, certain verbs require only sets or only individuals in certain argument positions. For example, the verb *meet* requires its agent to be a set with more than one element, as a single individual cannot meet. Other verbs require individuals and exclude sets, and others allow both sets and individuals as arguments.

Consider the different formulas that arise from the collective/distributive readings. There are two interpretations of the sentence *Some men bought a suit,* which has the following logical form (omitting the tense operator):

(SOME **m1** : (PLUR MAN1)
(A **s1** : SUIT1
(BUY1 **m1 s1**)))

The collective reading would map to

$$\exists M1 : M1 \subset \{z \mid Man(z)\}\ \exists s : Suit(s)\ .\ Buy1(M1, s)$$

that is, there is a subset of the set of all men who together bought a suit. The distributive reading involves some men individually buying suits and would be represented by

$$\exists M2 : M2 \subset \{z \mid Man(z)\}\ \forall m : m \in M2\ \ \exists s : Suit(s)\ .\ Buy1(m, s)$$

Note that the collective and distributive readings both involve a common core meaning involving the subset of men. The only difference is whether you use the set as a unit or quantify over all members of the set.

The set-based representation can also be used to ensure that more than one man bought a suit. To do this we introduce a new function that returns the cardinality of set. For any given set S, let $|S|$ be the number of elements in S. Using arithmetic operators, we can now encode constraints on the size of sets. For example, the meaning of *Three men entered the room* would be as follows, again with tense information omitted,

$$\exists M: (M \subset \{y \mid Man(y)\}\ \&\ |M| = 3)\ \ \forall m: m \in M\ .\ Enter1(m, Room1)$$

By changing the restriction to $|M| \geq 3$, you get the meaning of *At least three men entered the room,* and so on.

More problematic quantifiers can also be given an approximate meaning using sets. For instance, if we define *most* as being true if more than half of some set has a given property, then *Most men laughed* might have the meaning

$$\exists\, M : (M \subset \{\, y \mid Man(y)\} \;\&\; |M| \geq \frac{|\{\, y \mid Man(y)\}|}{2})$$
$$\forall\, m : m \in M \,.\, Laughed(m)$$

In an actual discourse, the interpretation of the quantified terms will usually be relative to some previously defined set. For example, the sentence *Most men laughed* typically will refer to most of the men in a previously mentioned set rather than to most of the men in the world. In other words, the sentence would not claim that more than half of all men laughed, but that more than half the men in a certain context (say in a given room) laughed. This type of interpretation will be discussed further in Chapter 14.

You have seen that by introducing sets as explicit objects in a representation, a wide range of quantificational constructs can be captured in an intuitively satisfying way. While the development here was in terms of extensions to FOPC, similar capabilities are needed in any representation to capture the same phenomena. For example, assume you are using a semantic network representation. To handle quantification you must be able to have nodes that represent sets, be able to state cardinality restrictions on these nodes, and be able to quantify over these sets to obtain the distributive reading.

○ 13.5 Time and Aspectual Classes of Verbs

One of the central components of any knowledge representation that supports natural language is the treatment of verbs and time. Much of language involves time, including temporal information implicit in the tense and aspect of sentences and explicit temporal information conveyed by a wide range of temporal adverbials (for example, *for five minutes, yesterday, at 3 o'clock, after they had left*).

In the logical form language, temporal information was handled in several ways. There were modal operators to represent tense (for example, PAST, PRES, PROG, FUT) and temporal connectives (for example, BEFORE, DURING), and all predicates could take time arguments. To handle such phenomena, we need to introduce additional extensions to FOPC to represent time.

There are several different types of times. A **time point** is an instantaneous time that is generally associated with some transition in the world, such as a light turning on or someone finding a lost pen. An **interval** of time is an extended stretch of time over which some event occurs. All intervals have **durations** (for example, five minutes long), while points cannot have durations. Many predicates can be defined only over intervals. For example, consider the predicate that asserts that John drove his car to work at a certain time. This can be true only over an interval of time, because driving to a destination necessarily takes time; you cannot drive in a single point.

Points and intervals have to be distinguished because different relationships can hold between them. For example, two intervals may overlap, whereas points cannot overlap. In addition, two intervals may **meet**: One ends where the other begins, but they do not overlap in time or have any time between them. A point or an interval may be contained within another interval, but nothing can be contained within a point. The following predicates are allowed for temporal relations:

$t1 < t2$	point/interval t1 is before point/interval t2
$t1 : t2$	interval t1 meets interval t2, or point t1 defines the beginning of interval t2, or point t2 defines the end of interval t1
$t1 \subseteq t2$	point/interval t1 is contained in interval t2

As previously mentioned, some predicates can be true only over intervals of times, whereas others can be true only at points, and others can be true at either. The classification of predicates corresponds with different aspectual classes of verb phrases.

Sentences describe propositions that fall into at least three distinct classes: those that describe states (**stative** propositions), those that define ongoing activities (**activity** propositions), and those that define completed events (**telic** propositions). Stative propositions describe some property of the world that can hold for an instant or extend indefinitely, as in the sentences

Jack is happy.
I believe the world is flat.

Stative propositions describe situations that lack a precisely defined ending point, and cannot appear in certain linguistic contexts. For instance, they do not naturally appear in the progressive form

*Jack is being happy.
*I am believing that the world is flat.

Activity propositions describe activities that occur over an interval of time. Activities are often expressed using the progressive form, as in the sentences

Jack is running.
The door was swinging to and fro.

Sentences describing states and activities do not usually allow temporal modifiers, such as *in five minutes,* but they do allow duration modifiers, such as *for five minutes.*

Telic sentences describe events that are brought to completion, as in

Jack fell asleep.
Jack climbed the mountain.

In both sentences, the event ends at some time (called the **culmination point**), and you know that some resulting property starts at the culmination point. For instance, with the first sentence you know that Jack is asleep at the end of the event, and with the second you know that Jack is at the top of the mountain.

Sentences describing telic propositions can include temporal modifiers such as *in an hour,* as in

> They climbed the mountain in two days.
> Jack fell asleep in an hour.

Telic eventualities are often broken down into two subclasses, depending on whether they essentially describe a transition only (the **achievement** class) or involve some activity leading up to the culmination (the **accomplishment** class). The previous examples describe accomplishments, whereas the following describe achievements:

> Jack recognized the man.
> Helen woke up.

The four types of proposition classes can be distinguished by different types of temporal arguments. In particular, stative propositions can be true at a point or an interval. For example, it makes sense to speak of a ball being red at a particular instant of time or over an extended interval of time. Stative propositions are **homogeneous**—whenever they hold over an interval, they also hold over all subintervals of that interval.

Achievement sentences, such as *Jack reached the summit* or *Helen closed the door,* map to propositions that describe transitions. The first describes a transition after which Jack is at the summit, whereas the second describes a transition after which the door is closed. Such predicates cannot hold over intervals, but their definitions might include information about resulting states, such as

$$\forall\ a1, l1, t1\ .\ Reach(a1, l1, t1) \supset \exists\ T1\ .\ t1 : T1\ \&\ At(a1, l1, T1)$$

that is, if an agent a1 reaches a location l1 at time point t1, then a1 is at l1 for some interval of time that starts at t1.

Propositions that describe processes correspond to activity verbs, such as *Jack ran.* Process predicates can occur only over intervals and tend to be homogeneous, although not in a strict way as with statives. In particular, it could be true that Jack was running between 2 and 3 o'clock, even if he stopped for a five-minute rest sometime during that time. Thus there can be a defeasible implication, but not an entailment, that if a process P occurs over an interval T1, then it is likely to have occurred over an interval T2 within T1.

Accomplishment sentences, such as *Jack ran to the store,* have a more complex structure that seems to combine several forms. We can handle this by mapping the logical form predicates to a more complex sentence in the KRL. In particular, the logical form for *Jack ran to the store* could map to a formula indicating that a process of running occurred that culminated in a state of being at the store, that is,

$$\exists\ T1, T2\ .\ Running(Jack1, T1)\ \&\ At(Jack1, Store1, T2)\ \&\ T1 : T2$$

Figure 13.4 summarizes some distinguishing properties of the aspectual classes.

Aspectual Class	Can Be True at a Point?	Can Be True at an Interval?	Temporal Modifier *in*
Stative Phrase	YES	YES	NO
Activity	NO	YES	NO
Achievement	YES	NO	YES
Accomplishment	NO	YES	YES

Figure 13.4 Different properties of the aspectual classes

Encoding Tense

Tense operators can also be represented directly in the temporal logic without the need for modal operators. The basic idea is to map tense operators to temporal relations with respect to some indexical term referring to the current time. For the following examples, let us assume that the constant NOW1 denotes the current time. Given this, we could map the PAST operator into a formula that existentially quantifies over a time before now; that is, the sentence *John was happy* would map to the KR expression

$$\exists\ T1\ .\ T1 < NOW1\ .\ Happy(Jack1, T1)$$

The same sentence in the simple present, *John is happy,* would map to

$$\exists\ T1\ .\ NOW1 \subseteq T1\ .\ Happy(John1, NOW1)$$

and the simple future, *John will be happy,* would map to

$$\exists\ T1\ .\ T1 > NOW1\ .\ Happy(Jack1, T1)$$

Note that there is ambiguity in the interpretation of tense, because some simple present sentences refer to the future, as in *The flight arrives at noon,* whereas some simple future sentences refer to the present, as in *Jack will be in class by now.* But we will ignore these complications in this development.

Even without ambiguity, there are some difficult problems. For instance, there are two ways to assert that something was true in the past, corresponding to the simple past and the past perfect, for example,

> Helen saw the books.
> Helen had seen the books.

What is the difference between these two readings? As isolated sentences, it is hard to tell, but consider these forms in more complex sentences, such as

> When Jack opened the door, Helen saw the books.
> When Jack opened the door, Helen had seen the books.

In the first sentence the act of seeing is cotemporal with or immediately after the time Jack opened the door, whereas in the second the act of seeing preceded

Jack's opening the door. The generally accepted account of this difference was proposed by Reichenbach (1947), who suggested the notion of **reference time**. In these examples the reference time for the main clause is the time that Jack opened the door. The simple past equates the time of the event (of seeing) with the reference time, whereas the past perfect asserts that the event precedes the reference time. Specifically, Reichenbach developed a theory that tense gives information about three times:

S — the time of speech
E — the time of the event/state
R — the reference time

The reference time can be provided by temporal adverbials, as above, or can often be determined by the discourse context, as will be discussed in Chapter 15.

In the simple tenses the reference time is the same as the event time, that is, E = R. The three forms are generated by varying the relationship between R and S:

Jack sings	simple present: S = R, E = R
Jack sang	simple past: R < S, E = R
Jack will sing	simple future: S < R, E = R

The perfect tenses, on the other hand, have the event time preceding the reference time, and differ, as before, in terms of how the reference time and speech time are related. Thus we have

Jack has sung	present perfect: S = R, E < R
Jack had sung	past perfect: R < S, E < R
Jack will have sung	future perfect: S < R, E < R

This analysis also provides an account of the posterior tenses, in which R < E:

Jack is going to sing	posterior present: S = R, R < E
Jack was going to sing	posterior past: R < S, R < E
Jack will be going to sing	posterior future: S < R, R < E

These orderings are shown graphically in Figure 13.5.

○ 13.6 Automating Deduction in Logic-Based Representations

The previous sections have developed the abstract knowledge representation language that will be used through the rest of this book. The remaining sections change the focus and look at selected reasoning strategies used in knowledge representation systems. This section describes some techniques for automated reasoning in knowledge representations. As mentioned earlier, reasoning systems fall into two main categories: the declarative techniques based on deductive proof techniques and the procedural approaches. This section considers some purely deductive techniques.

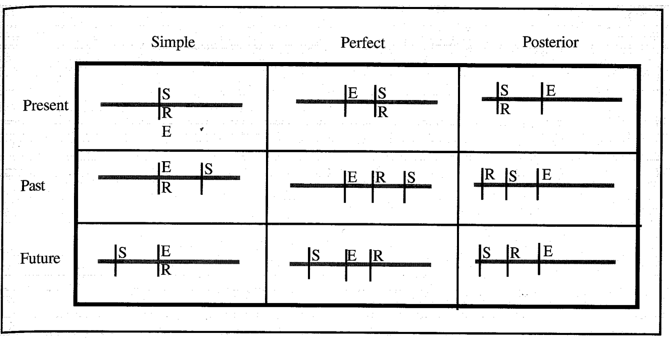

Figure 13.5 Some temporal configurations allowed by the tenses

If you have ever studied proof methods such as natural deduction for FOPC, you know that finding proofs is complicated because there are many different ways to try to prove any given formula. In addition, there are often many syntactic ways to express the same content. Most work in automated reasoning attempts to reduce this complexity before starting the task. In particular, a **normal form** is used for formulas that uses a restricted subset of the full FOPC syntax. Many formulas that are logically equivalent but syntactically different have the same normal form. For example, the formulas

$\neg P \,\&\, Q$
$\neg(P \vee \neg Q)$
$\neg(Q \supset P)$

are all logically equivalent. In a representation based on **conjunctive normal form** (CNF), described later in this section, these all map to the same formula. A second technique concerns the handling of variables. In general, with quantified variables, many particular instantiations of those variables could be used to establish a proof. Thus, when searching for a proof, there is ample opportunity to pick the wrong instantiation and have to reconsider later. The **unification** technique partially avoids this problem by always computing the most general solution possible for a given line of reasoning. If you are not already familiar with unification, see Appendices A and B.

Many different reasoning systems can be viewed with the framework of automatic proof systems, from simple pattern-based retrieval from databases, to Horn clause systems similar to PROLOG, to fully general theorem-proving systems. The differences arise from the form of expressions each system can represent and reason about. To see this, let's first develop a fully general normal form for FOPC. At the level of constants, functions, and atomic propositions,

Formula in FOPC	Clause Form Equivalent
P	(*P* <-)
$\neg P$	(<- *P*)
$P \,\&\, Q$	two clauses: (*P* <-) and (*Q* <-)
$P \vee Q$	(*P Q* <-)
$Q \vee \neg P$	(*Q* <- *P*)
$P \supset Q$	(*Q* <- *P*)
$\neg(P \supset Q)$	two clauses: (*P* <-) and (<- *Q*)
$(P \,\&\, Q) \vee R$	two clauses: (*P R* <-) and (*Q R* <-)

Figure 13.6 Formulas in clause form

conjunctive normal form is identical to FOPC. A **literal** corresponds to an atomic proposition, possibly negated, such as the following:

Person(*John1*)
Car(*Car1*)
Owns(*John1, Car1*)
¬*Happy*(*John1*)

A **clause** is simply a disjunction of literals such as the following, which asserts that either Helen owns a particular car or she is not happy:

Owns(*Helen1, Car2*) ∨ ¬*Happy*(*Helen1*)

In many systems, clauses are written using a form of the implication operator instead, and the preceding clause would be written as

(*Owns*(*Helen1, Car2*) <- *Happy*(*Helen1*))

In general, a clause is written as

(*P1*, ..., *Pn* <- *Q1*, ..., *Qm*)

which states that if *Q1*, ..., *Qm* are true, then at least one of *P1*, ..., *Pn* is true. Viewing a clause as a disjunctive formula, the *Qi*'s are all the negated literals, while the *Pi*'s are all the positive literals. It can be shown that all formulas in FOPC have an equivalent clause form. Figure 13.6 gives some examples of some propositional formulas using standard logical operators and their equivalent form in conjunctive normal form. Quantification is handled in clause-based systems using variables and skolemization, as discussed in Section 13.2.

A very useful restriction of this type of expression is the **Horn clause**, which has exactly one literal on the left side, that is, exactly one positive literal. In a KB restricted to Horn clauses, the backward chaining strategy used in PROLOG is able to find a proof of any formula that logically follows from the KB.

Reasoning with clauses makes extensive use of the unification algorithm. Consider first a very limited KR that allows only literals in the KB, similar to a

relational database that allows quantification. The intuition we want is that a literal P follows from the KB if we can retrieve a formula in the KB that unifies with it. Consider how variables are treated in such a system. The FOPC formula ∀ *y* ∃ *x P*(*x, y*) would correspond to the literal *P*(*Sk2*(*?y*), *?y*). If the KB contains this literal, consider what would happen if you later want to see whether ∀ *w* ∃ *z P*(*w, z*) follows from the KB. If you convert it to clause form, you would obtain a literal such as *P*(*Sk3*(*?w*), *?w*), which will not unify with the literal in the database since the Skolem functions are named differently. In pattern retrieval systems, this is handled by changing the interpretation of quantifiers when they are used as a query. Thus, as a query, ∀ *y* ∃ *x P*(*x, y*) would map to a literal *P*(*?z, Sk4*), which would unify with the literal for the same formula already in the knowledge base.

At first glance, this technique of changing the interpretation of quantifiers for queries may seem rather arbitrary, but it actually follows from the underlying proof strategy being used. In particular, there are always two general methods for proving a formula P. The first is to build a proof of P directly from the KB using the rules of inference. The second is to show that ¬P is inconsistent with the KB (and thus P must follow from the KB). The latter approach is called a **refutation proof** and turns out to be the most useful technique for automatic deduction. It forms the foundation for pattern-matching techniques as previously described, Horn clause proof strategies, and general resolution-based theorem-proving systems. All of these can be viewed as specialized implementations of a single rule of inference called the **resolution rule**. A simple case of the resolution rule resembles modus ponens. In particular, given a clause

(*Q* <- *P*) (that is, *P* implies *Q*)

and the clause

(*P* <-) (that is, *P* is true)

the resolution rule allows you to conclude (*Q* <-) (that is, *Q* is true). Another simple case of the resolution rule detects contradictions. Given (*P* <-) (that is, *P* is true) and (<- *P*) (that is, *P* is false), the resolution rule gives the empty clause (<-), which indicates that the database is inconsistent.

The resolution rule is generalized to FOPC by using unification to instantiate the variables in the two clauses to make the X's identical. For example, consider a KB that includes clauses asserting that all dogs bark and that Fido is a dog:

(*Bark*(*?x*) <- *Dog*(*?x*))
(*Dog*(*Fido1*) <-)

The resolution rule would allow you to conclude that Fido barks, for after substituting *Fido1* for *?x* in the two clauses, you can cancel the literal *Dog*(*Fido1*) from both clauses to obtain the resulting clause:

(*Bark*(*Fido1*) <-)

With the resolution rule in hand, you can now consider the refutation proof strategy. Given a consistent KB in clause form, we can determine whether a formula P follows from the KB by negating *P*, converting it to clause form and adding it to the KB, and then showing that we can derive the empty clause using the resolution rule. Since this indicates that the KB is now inconsistent, *P* must follow from the KB. It can be proven that the resolution strategy is complete in the sense that if *P* does follow from the KB, then a proof can be found. However, the converse is not true in the general case. If *P* does not follow from the KB, the proof strategy may never be able to tell this fact. By limiting the form of the clauses that can be used, you obtain different properties. For instance, with a KB consisting solely of Horn clauses, you can tell whether a formula *P* does or does not follow from the KB in every circumstance.

In deductively based systems, a common technique for introducing a default mechanism is called **proof by failure**. A new operator called UNLESS is introduced that recursively calls the theorem prover on the formula that is its argument. If the recursive call to the theorem prover stops without proving the formula true, then the UNLESS formula is true. For example, the default rule that cats purr might be expressed as the following axiom:

$$\forall\, c\,.\, Cat(c)\ \&\ Unless(\neg Purr(c)) \supset Purr(c)$$

That is, you can conclude that a cat purrs except when you can prove it doesn't purr. Because of the potential expense of recursively calling the prover, such techniques are usually used only with restricted proof systems, such as in PROLOG-style Horn-clause representations. Another technique used in many such systems that is closely related to the closed world assumption is the **negation as failure** rule, where a proposition is false if it can't be proven true; that is, for any proposition *P*, $Unless(P) \supset \neg P$. Of course, you would have to be very careful if using default rules in a system that uses negation as failure. For instance, in a system using negation as failure, the previous default rule would state that you can conclude that cats purr only when you can conclude that cats purr—not a very useful rule!

The notions of clauses, unification, and refutation proofs provide the formal underpinnings of virtually every modern knowledge representation system; that is, any system that uses pattern matching with variables can be seen as a special case of the general technique. Of course, this does not mean that matching covers all the reasoning that a knowledge representation system can do, but it is a crucial part of every system.

○ 13.7 Procedural Semantics and Question Answering

Procedurally based techniques are frequently used in database query applications, where there is a large difference in expressive power between the logical form language and the database language. Cast in terms of the formalism in the last section, the KB (that is, the database) consists only of positive literals, often

(FLIGHT F1)	(ATIME F2 CHI 1000HR)
(FLIGHT F2)	(ATIME F3 CHI 900HR)
(FLIGHT F3)	(ATIME F4 BOS 1700HR)
(FLIGHT F4)	(DTIME F1 BOS 1600HR)
(AIRPORT BOS)	(DTIME F2 BOS 900HR)
(AIRPORT CHI)	(DTIME F3 BOS 800HR)
(ATIME F1 CHI 1700HR)	(DTIME F4 CHI 1600HR)

Figure 13.7 A simple database of airline schedules

without variables. Rather than convert the logical form language into extended FOPC as described in earlier sections, the logical forms are treated as expressions in a query language. Each logical form language construct corresponds to a particular procedure that performs the appropriate query. For example, the query *Does every flight to Chicago serve breakfast?* with the logical form

(EVERY **f1** : (& (FLIGHT **f1**) (DEST **f1** (NAME **c1** "Chicago")))
(SERVE-BREAKFAST **f1**))

would be interpreted as a procedure as follows:

1. Find all flights in the database with destination CHI (the database symbol for Chicago).
2. For each flight found, check if it serves breakfast. If all do, return yes; otherwise return no.

This section shows how to interpret logical form expressions as procedures, a method of interpretation often called **procedural semantics**.

To make the development concrete, consider the very simple database retrieval system shown in Figure 13.7. The database consists of a set of positive literals containing no variables. Times are indicated in international notation; for example, 1700HR is 5:00 PM. The relation (ATIME f c t) indicates that flight f arrives at airport c at time t, and (DTIME f c t) indicates that flight f leaves from airport c at time t. The database system provides a simple interface based on pattern matching of literals, where the query may contain variables. Two database query functions are assumed:

(Test <literal>$_1$ ..., <literal>$_n$)—returns true if there is some binding of the variables such that each literal is found in the database.

(Retrieve <var> <literal>$_1$ ..., <literal>$_n$)—like Test, but if it succeeds it returns every instance of the indicated variable that provides a solution.

For example, given the database in Figure 13.7, the query

(Retrieve ?x (FLIGHT ?x) (ATIME ?x CHI 1000HR)) would return the list (F2) because F2 is the only binding of ?x where both these literals are in the database.

All expressions in the logical form language must be interpreted in a way that reduces eventually to these two query forms on the database. The way this is done is by mapping the logical form into a procedure that performs the appropriate queries on the database. Thus answering a question is done in two steps: translating the logical form into a program and then executing that program to compute the answer.

Consider the translation step first. For any logical form expression E, the translation of E in the database query language will be indicated as T(E). The translation of expressions varies depending on the constructs. For instance, expressions such as (NAME **c1** "Chicago") will be translated into the appropriate database constant, in this case CHI. But in addition, the symbol **c1** must be stored with the constant CHI on a structure called the **symbol table**, so that if **c1** is found again in another part of the logical form, it can also be replaced by its value CHI.

Some logical form relations will translate directly into database relations, whereas others will translate into more complex expressions. For instance, the logical form relation DEST is not used in the database; rather, the destination of a flight is encoded in the ATIME relation that includes both the flight's destination and its arrival time. Thus the logical form relation (DEST **f1** (NAME **c1** "Chicago")), where **f1** has already been associated with a variable ?f, would translate into the database relation

(ATIME ?f CHI ?t)

Since the time is not included in the DEST relation, it is interpreted as an unconstrained variable in the translation. In general, the translation of each relation in the logical form must be specified.

The procedural semantics approach gets more interesting as it interprets logical connectives and quantifiers, which of course have no corresponding constructs in the relational database. The logical operators are interpreted as follows:

Conjunctions: (& $\mathbf{R_1}$ ..., $\mathbf{R_n}$)—will translate into a program of the form (CHECK-ALL-TRUE T(R_1) ..., T(R_n)), which when executed will successively query each T(R_i) to make sure it is true and pass on the variable bindings to the queries that follow. If there is a set of variable bindings such that querying each T(R_i) succeeds, then the program succeeds; otherwise it fails.

Disjunctions: (OR $\mathbf{R_1}$ ..., $\mathbf{R_n}$)—will translate into a program of the form (FIND-ONE-TRUE T(R_1) ..., T(R_n)), which when executed will successively query each T(R_i) until one of the R_i succeeds, in which case the program succeeds. If no R_i succeeds, then the program fails.

The procedure for negation assumes the closed world assumption on all relations in the database, and uses proof by failure:

(NOT R)—translates into a program of the form (UNLESS T(R)), which succeeds only if querying T(R) fails.

The most complex translations occur with quantifiers. Each quantifier translates to a program that does the appropriate operations on the database. Because of the limitations of the database language, only the distributive readings of plural quantifiers are usually supported. Consider three quantifiers important in question-answering applications: THE, EACH and WH.

(THE x : $\mathbf{R_x}$ $\mathbf{P_x}$)—translates into a program (FIND-THE ?x T($R_{?x}$) T($P_{?x}$)), which first does a retrieval to find all ?x that satisfy T($R_{?x}$), that is, (Retrieve ?x T($R_{?x}$)). If a single answer is found, then that answer is substituted for ?x in the entire expression, and T($P_{?x}$) is executed to provide the answer for the entire expression. If no object is found when querying T($R_{?x}$), then there is a presupposition violation that might be handled by the question-answering system in a special way, say, notifying the user that there is no such object. If multiple answers are found, the designer of the system must decide what is best to do. Some systems allow this situation and execute T(P_x) for each of the values; other systems treat it as a failure.

(EACH x : $\mathbf{R_x}$ $\mathbf{P_x}$)—translates to a program (ITERATE ?x T($R_{?x}$) T($P_{?x}$)), which also starts by doing a retrieval to find all ?x that satisfy T($R_{?x}$). It then iteratively executes T($P_{?x}$) for each value found and succeeds only if each of these queries succeeds.

(WH x : $\mathbf{R_x}$ $\mathbf{P_x}$)—translates into a program (PRINT-ALL ?x T($R_{?x}$) T($P_{?x}$)), which retrieves all objects that satisfy the translations of R_x and P_x, that is, (Retrieve ?x T($R_{?x}$) T($P_{?x}$)), and then prints out the results. Determining the best format for printing the answers, especially determining whether additional information should be provided, is a complex issue. Here we assume it simply prints the answers found.

This is enough mechanism to show some examples using the database in Figure 13.7. The query *Which flight to Chicago leaves at 4PM?* would have the logical form (after scoping)

(WH **f1** : (& (FLIGHT **f1**) (DEST **f1** (NAME **c1** "Chicago")))
    (LEAVE **l1** (NAME **t1** "4PM")))

This would translate into a query of the form

(PRINT-ALL ?f (FLIGHT ?f) (ATIME ?f CHI ?t) (DTIME ?f ?s 1600HR))

Here, the DEST relation maps to an ATIME relation as previously described, and the LEAVE predicate maps into the DTIME relations. Note that the departure location was not specified in the logical form and so is treated as a variable here. In a real application the departure city would be determined by context or by default. With the small database shown in Figure 13.7, however, there is only one flight matching the current description, namely F1, so it works in this case.

Consider a more complex example that involves iteration, as in the request *Give the departure time of each flight to Chicago* with the logical form

```
(EACH f1 : (& (FLIGHT f1) (DEST f1 (NAME c1 "Chicago")))
      (THE t1 : (DEPART-TIME f1 t1)
            (GIVE-SPECIFY1 g1)))
```

This would translate into the query

```
(ITERATE ?f1 (CHECK-ALL-TRUE (FLIGHT ?f1) (ATIME ?f CHI ?t1))
      (FIND-THE ?t1 (DTIME ?f1 ?city ?t1)
            (PRINT ?t1)))
```

In this case, the interpretation of the verb *give* simply involves printing out its argument, i.e., the departure time. The execution of the expression then proceeds as follows:

1. The first part of the ITERATE step is to find all ?f1 satisfying the restriction. The CHECK-ALL-TRUE procedure succeeds for ?f1 only if both (FLIGHT ?f1) and (ATIME ?f CHI ?t1) are in the database. This step returns the flights F1, F2, and F3.
2. The second part of the step is to execute (FIND-THE ?t1 (DTIME ?f1 ?city ?t1) (PRINT ?t1)) for each of the three values. Consider the execution with the first value, F1. The expression is

   ```
   (FIND-THE ?t1 (DTIME F1 ?city ?t1) (PRINT ?t1))
   ```

   The program for FIND-THE first performs the query (Retrieve ?t1 (DTIME F1 ?city ?t1)). This returns a unique answer, namely the time 1700HR. The second step of the FIND-THE program executes PRINT on this value, causing 1700HR to be printed. The second and third iterations print the values 1000HR and 900HR, respectively.

Many natural language database query systems use the procedural semantics technique. It provides a convenient way to capture the appropriate behavior for many constructs whose meaning cannot be expressed within the limited language of the database system. Because of the nature of database applications, the limitations of these techniques do not appear to be a problem in practice. For instance, database systems don't typically encode information that would make queries using collective interpretations of quantifiers necessary. In addition, since the database does not contain disjunctive information, the limited forms for disjunctive queries also do not pose a problem.

BOX 13.3 LUNAR: A Natural Language Database Query System

With the discussion of procedural semantics, you have now seen most of the central components of the LUNAR system. LUNAR, developed in the 1970s, acted as a front-end query system to a database containing information about the rock samples brought back from the Apollo missions to the moon. It was the first natural language system to demonstrate extensive coverage in a realistic application domain, and many of the techniques that are common in the field today either originated or were first developed to an advanced stage in this system. The system used an ATN parser (see Section 4.6) that produced a representation based on grammatical relations, which was then interpreted by a semantic interpretation module that used a recursive pattern-matching technique to produce an expression in a meaning representation, as in Section 11.1. Quantifier information was maintained separately from the rest of the semantic representation and was then ordered using heuristics similar to those described in Section 12.3. The result was a final meaning representation expressed in a meaning representation language similar to our fully scoped logical form. This was then executed using a procedural semantics approach as described in this section. Some examples of queries that LUNAR could handle are

Give me all lunar samples with magnetite.
In which samples has apatite been identified?
What is the specific activity of A126 in soil?
What is the average concentration of olivine in brecchias?
In which brecchias is the average concentration of titanium greater than 6 percent?

For more information on LUNAR, see Woods (1970; 1977; 1978).

Procedural semantic techniques can also be used with Horn-clause-based databases as well. Most of the procedural definitions of constructs can be defined by Horn clause axioms, with the addition of an ability to recursively invoke the prover to perform tasks such as finding all objects that satisfy some set of literals. With extensions to handle finite sets, such a representation can handle a wide range of quantifiers procedurally using the encoding techniques described in Section 13.4. With their additional expressive power, Horn-clause-based databases are a very attractive generalization to the traditional relational database for supporting natural language query systems.

○ 13.8 Hybrid Knowledge Representations

Even if a knowledge representation language remained first-order, general search strategies in theorem proving would usually be too inefficient for practical systems. The theoretical cleanness of viewing inference as theorem proving,

$$\begin{tikzcd}
\text{ANIMAL} \\
\text{MAMMAL} \arrow[u, "s"'] \\
\text{DOG} \arrow[u, "s"'] \\
\text{Fido1} \arrow[u, "isa"']
\end{tikzcd}$$

Figure 13.8 A small type hierarchy

however, has many attractive properties. Hybrid KR systems attempt to gain the advantages of using efficient procedural inference for some tasks while retaining the theoretical framework of theorem-proving systems.

As a start, the ideas of unification and refutation proof can be carried over into most systems. A hybrid system, however, does not depend entirely on these techniques. Rather, certain forms of inference are accomplished using special-purpose techniques that can be considerably more efficient. For example, consider the implementation of type hierarchies in a KR system. You saw earlier that type hierarchies can be encoded as axioms (for example, $\forall$ x . DOG(x) $\supset$ MAMMAL(x)), or as graphs, as in semantic networks. These techniques may be formally equivalent, but they can produce radically different computational properties.

One way to combine these techniques is to assume a typed logic resembling the restricted quantification logic developed in Section 13.2. The type hierarchy is predefined in a semantic network structure in which DOG is a subtype of MAMMAL which is a subtype of ANIMAL, and Fido1 is predefined to be a member of the set DOG. Given this general knowledge, the KB encoding the assertion *All animals have a mother* would be

(*MOTHER*(*?x:ANIMAL*, *Sk1*(*?x*)) <-)

where the notation *?x:ANIMAL* indicates a variable ranging over type *ANIMAL*. Such expressions could be reasoned about by extending the unification algorithm, so that two terms may unify only if they are of compatible types; that is, *?x:ANIMAL* and *Fido1* will unify only if *Fido1* is a member of the set *ANIMAL*. This constraint can be checked procedurally using the semantic network shown in Figure 13.8. Now the query as to whether Fido has a mother—that is, *MOTHER*(*Fido1, ?y*)—can be proved using a single unification step.

The procedural approach allows you to write highly optimized procedures that are significantly faster than would be possible doing the same work using axioms. The hybrid representation also allows for a more intuitive encoding of the information, using a semantic network for the type information, and also could permit other nondeductive algorithms to be performed on the semantic network.

Another example of a specialized reasoner that can be put to very effective use concerns equality reasoning. It is very difficult to axiomatize equality directly into a theorem-proving system because it is hard to encode the equivalence of formulas that differ only in using two different names for the same object. Very efficient algorithms exist, however, for maintaining equality information between ground terms based on equivalence classes. If such techniques are built into the unifier, then no explicit axioms about equality need be encoded in the system. Rather, the extended unification algorithm would use the procedures defined for equality to check whether two terms are equal and thus can be unified.

Other forms of specialized reasoning systems can be integrated by defining procedures that establish the truth of particular predicates. The technique is called **procedural attachment**. For instance, consider temporal reasoning. While it is possible to use an axiomatization of time to drive temporal reasoning, the resulting system would be very inefficient. There are, however, specialized reasoning techniques that can manage temporal information quite effectively. Such systems can be integrated into a hybrid system using special predicates. To see this, consider what roles propositions play in a reasoning system. There are generally three different operations applicable to propositions:

Assert that it is true (that is, add it to the KB)
Query whether it is true (that is, invoke the theorem prover on it)
Retract it (that is, remove it from the KB)

While each of these operations was defined in terms of a theorem-proving system, this does not have to be the only way such operations are accomplished. In fact, you could define arbitrary procedures to perform each of the tasks. For instance, consider a specialized temporal reasoning system that maintains a graph of temporal relations and uses graph search techniques to establish temporal relations. Assume that there is a predicate BEFORE in the KB that indicates that one time precedes another. When a proposition such as (BEFORE t1 t2) is to be added to the KB, the specialized temporal reasoner is invoked to add the information to its temporal graph. When the same proposition is queried (either directly by the user, or as a substep of a more complex proof), then the specialized temporal reasoner is called to establish it. As a result, the specialized temporal reasoner can be fully integrated into the theorem prover, and used whenever temporal information is required. Of course, to be fully integrated, the specialized reasoning would have to be able to handle variables and return results equivalent to unification. For instance, if the theorem prover needed to establish (BEFORE t1 ?x), the temporal reasoner would have to return a binding for ?x. In addition, it would need to be able to handle backtracking when alternate solutions need to be explored.

Hybrid reasoning systems offer an attractive way to integrate specialized reasoning algorithms into a uniform framework.

Summary

Natural language understanding requires a capability to represent and reason about knowledge of the world. While there are many different techniques for representing knowledge, every representation sufficient for general language understanding must at least support the following general capabilities:

- a full range of logical operators and logical quantification, as found in FOPC
- a way to represent default, stereotypical information about the objects and situations that occur in the domain
- a way to explicitly represent and reason about finite sets
- a method of representing and reasoning about temporal information

These are representative but by no means exhaust the areas of concern. In a fully general system, for instance, you would also explore the spatial information in language, and the representation of mental attitudes.

This chapter developed an abstract representation language, combining the techniques of FOPC and frame-based systems, that satisfies the requirements just listed. This abstract representation could be realized within a wide range of knowledge representation systems using different techniques. Any knowledge representation system must support basic capabilities for pattern matching, for which the notion of unification and inference based on refutation provide a formal basis. Many systems also specify specialized procedures for some or all of the reasoning tasks. A system that uses a mix of techniques, including deductive and procedural techniques, is called a hybrid system.

Related Work and Further Readings

Knowledge representation is a highly diverse area in artificial intelligence and is fundamental to many problems beyond language understanding. A good introduction to the field is the collection of papers in Brachman and Levesque (1985). A good sample of current work in the field can be found in the proceedings of the Conferences on Knowledge Representation and Reasoning (for example, Brachman, Levesque, and Reiter (1989); Allen, Fikes, and Sandewall (1991); and Nebel, Rich, and Swartout (1992)). Norvig (1992) has written an excellent text that discusses different implementation techniques for knowledge representation systems.

The introduction of frames by Minsky (1975) produced a large body of subsequent work in representation. One of the first knowledge representation systems based on these ideas was KRL (Bobrow and Winograd, 1977). Hayes (1979) performed an analysis of KRL in terms of FOPC. Most modern representation systems, such as the systems described in Brachman and Levesque (1985), can be seen as combinations of frame systems, semantic networks, and deductive

logic. A large class of systems organize knowledge around descriptions of categories of objects and are called **term subsumption languages.** There are good examples in Brachman and Levesque (1985). A good reference for semantic network-based systems is Sowa (1991).

The strongest proponents of knowledge representations based on decompositions into a small set of primitives have been Schank (1975) and Wilks (1975). Most current representation systems, however, use abstraction as the organizational tools in representations, and are often based on semantic networks and frame systems. There have also been many proposals for decomposition in linguistics (such as Dowty (1979) and Jackendoff (1990)).

The treatment of quantifiers by using explicit sets is common in computational systems (for example, Woods (1977); Warren and Pereira (1982); and Alshawi (1992)) and in linguistics (for example, McCawley (1993)).

The study of tense and aspect is a very active area of research in linguistics, philosophy, and computational linguistics. A good reference on tense and aspect in the computational literature is a special issue of *Computational Linguistics* (1988). An excellent place to start in the linguistics literature is with Dowty (1979; 1986) and Bach (1986). More recent work in the area includes Parsons (1990) and Pustejovsky (1991). The classic reference for tense is Reichenbach (1947). There is also a large literature on the treatment of tense as a modal operator (for example, Prior (1967)). McCawley (1993) contains a good introduction to linguistic issues in dealing with tense and aspect. Allen (1984) describes a temporal logic that explicitly involves predicates in three different aspectual categories. Davis (1990) describes a logic-based representation that includes specialized representations for time, space, and many other aspects of the world.

Most deductive techniques have evolved from work in resolution theory proving as introduced by Robinson (1965). Robinson introduced the resolution rule and proved that the resolution refutation proof technique was complete. The technique of proof by failure was used in the early AI programming language PLANNER (Hewitt, 1971), and was formalized by Clark (1978). Much of the work on default logics stems from work by Reiter (1980). Etherington and Reiter (1983) used this formalism to define inheritance formally with exceptions in type hierarchies. There is a large body of literature on representation using semantic techniques based on minimal models. A good general overview can be found in Genesereth and Nilsson (1987).

Procedural semantics was extensively used in early systems, notably Winograd (1973) and Woods (1978), and is still a common technique in database query systems. The CHAT-80 system (Warren and Pereira, 1982) used similar techniques within a PROLOG-based representation to produce an elegant and quite powerful query mechanism. For a brief survey of question-answering techniques, see Webber (1992). A good example of a more current question-answering system is the TEAM system (Grosz et al., 1987). TEAM was aimed at being transportable, which means it can be relatively easily adapted to a different database without having to rewrite the grammar and semantic interpreter. To do this, it uses a context-independent logical form similar to the approach used in Chapter 8.

Exercises for Chapter 13

1. (*easy*) Give a plausible logic-based representation for the meaning of the following sentences, focusing on the interpretation of the quantifiers. If the sentence has a collective/distributive ambiguity, give both interpretations.

Several men cried.
Seven men in the book met in the park.
All but three men bought a suit.

2. (*easy*) For each of the following lists of sentences, state whether the first sentence entails or implies each of the sentences that follow it, or that there is no semantic relationship between them. Justify your answers using some linguistics tests.

a. John didn't manage to find the key.
John didn't find the key.
John looked for the key.
The key is hard to find.

b. John was disappointed that Fido was last in the dog show.
Fido was last in the dog show.
Fido was entered in the dog show.
John wanted Fido to win.
Fido is a stupid dog.

3. (*easy*) Classify the indicated verb phrases in the following sentences as to whether they describe a state, an activity, an achievement, or an accomplishment. Justify your answers with some examples that demonstrate their linguistic behavior. Discuss any problems that arise in your classification.

Jack *ran to the store.*
Jack *was running to the store.*
Jack *hated running.*
Jack *runs* every day.
Jack *stopped running* when he broke his leg.

4. (*medium*) One of the classic examples of decompositional semantics is the encoding of the verb *kill* using a causation operator and a predicate DIE. In particular, the meaning of the sentence *John killed Sam* would be

CAUSE(John1, DIE1(Sam))

Does this decomposition completely capture the meaning of the verb *kill*? Consider whether *John killed Sam* and *John caused Sam to die* are equivalent in all situations. Given your position on this issue, would it be better for a KR to decompose all instances of *kill* to this form, or to use a meaning postulate and retain a predicate KILL in the KR? Justify your answer.

5. (*medium*) Using the translation of inheritance networks into logic described in Section 13.1, give the axioms for the simple network

Note that the definition of the drive action places a more restrictive constraint on the type of the THEME role. Does the axiomatization do the right thing, that is, does it show that the resulting axioms are consistent and that, for any drive action D, ∃ o . THEME(D,o) ∧ CAR(o)? Did the definition of the THEME role on the general class OBJ/ACTION interfere with this in any way? In answering these questions, explicitly specify any assumptions you need to make about the type hierarchy to do each proof.

6. (*medium*) Using the frame-based representation described in this chapter, define an action class DRIVE that corresponds to a sense of the verb *drive* in

i. I drove to school today.

In particular, your definition should contain enough detail so that each of the following statements could be concluded from sentence i.

ii. I was inside the car at some time.
iii. I had the car keys.
iv. The car was at school for some time.
v. I opened the car door.

For each of these sentences, discuss in detail how the necessary knowledge is represented (as a precondition, effect, decomposition, and so on) and what general principle justifies it being a conclusion of sentence i. Identify three other conclusions that can be made from sentence i, and discuss how the required knowledge to make each conclusion is encoded. Should any of the definitions be considered to be default knowledge? If so, why? If not, identify one further conclusion that could be made if some default knowledge were used.