neural structures. For attempts to simulate the brain at a physiological level, see the books by Albus (1981), E. Kent (1981), and de Callatay (forthcoming). Hinton and Anderson (1981) edited a collection of papers on holographic and associative models. Those approaches begin with physiological evidence, while the present book starts with linguistics, psychology, and philosophy.

Seminarium 1

Excerpted from
"Conceptual Structures"
by John Sowa

with compliment
25/1/2012

3 Conceptual Graphs

Conceptual graphs form a knowledge representation language based on linguistics, psychology, and philosophy. In the graphs, concept nodes represent entities, attributes, states, and events, and relation nodes show how the concepts are interconnected. This chapter defines the graphs formally, and the next chapter applies them to logic and computation.

3.1 PERCEPTS AND CONCEPTS

Perception is the process of building a *working model* that represents and interprets sensory input. The model has two components: a sensory part formed from a mosaic of *percepts*, each of which matches some aspect of the input; and a more abstract part called a *conceptual graph*, which describes how the percepts fit together to form the mosaic. Perception is based on the following mechanisms:

- Stimulation is recorded for a fraction of a second in a form called a *sensory icon.*
- The *associative comparator* searches long-term memory for percepts that match all or part of an icon.
- The *assembler* puts the percepts together in a working model that forms a close approximation to the input. A record of the assembly is stored as a conceptual graph.
- Conceptual mechanisms process *concrete concepts* that have associated percepts and *abstract concepts* that do not have any associated percepts.

When a person sees a cat, light waves reflected from the cat are received as a sensory icon *s*. The associative comparator matches *s* either to a single cat percept *p* or to a collection of percepts, which are combined by the assembler into a complete image. As the assembler combines percepts, it records the percepts and their intercon-

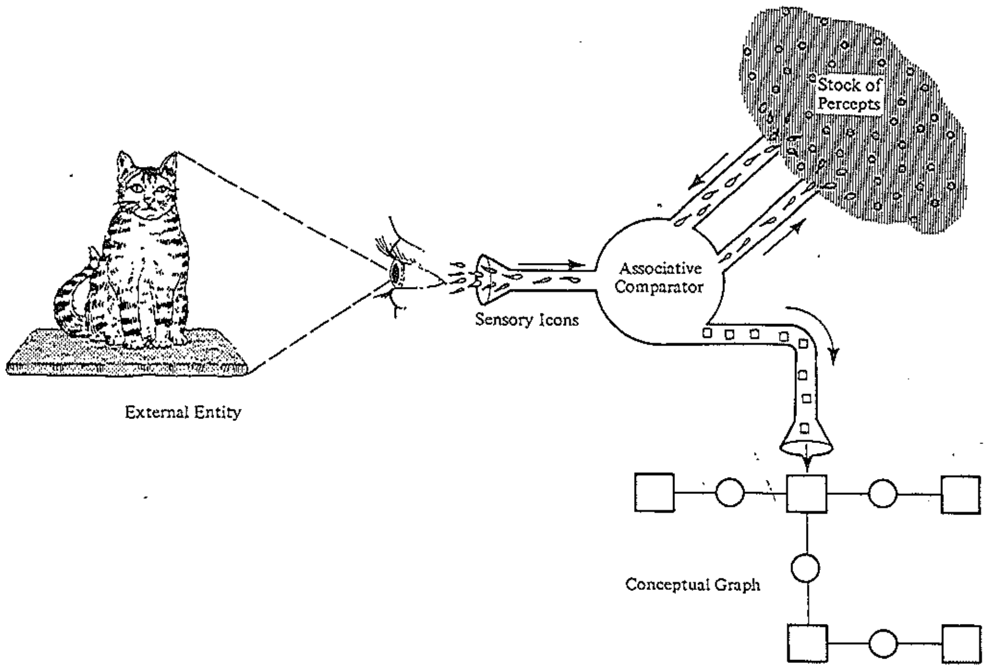

Fig. 3.1 The process of perceiving a cat

nections in a conceptual graph (Fig. 3.1). In diagrams, conceptual graphs are drawn as linked boxes and circles. Those links represent logical associations in the brain, not the actual shapes of the neural excitations.

3.1.1 Assumption. The process of *perception* generates a structure u called a *conceptual graph* in response to some external entity or scene e:

- The entity e gives rise to a *sensory icon* s.
- The *associative comparator* finds one or more *percepts* $p_1,...,p_n$ that match all or part of s.
- The *assembler* combines the percepts $p_1,...,p_n$ to form a *working model* that approximates s.
- If such a working model can be constructed, the entity e is said to be *recognized* by the percepts $p_1,...,p_n$.
- For each percept p_i in the working model, there is a *concept* c_i called the *interpretation* of p_i.
- The concepts $c_1,...,c_n$ are linked by *conceptual relations* to form the conceptual graph u.

Percepts are fragments of images that fit together like the pieces of a jigsaw puzzle. A conceptual graph describes the way percepts are assembled. Conceptual relations specify the *role* that each percept plays: one percept may match a part of an icon to the right or left of another percept; a percept for a color may be combined with a percept of a shape to form a graph that represents a colored shape. For auditory percepts, a graph may specify how percepts for phonemes are assembled to form syllables and words. Such graphs have been used for both language and vision. Winston (1975a) wrote a program based on graphs that learns patterns for various structures. Given graphs for arches and nonarches or pedestals and nonpedestals, his program analyzed the graphs in order to determine the conditions for a structure to be called an arch or a pedestal.

Figure 3.2 (patterned after Winston 1975a, p. 198) shows a conceptual graph that describes a simple arch consisting of two standing bricks and an arbitrary object lying across the bricks. The boxes in the diagram represent concepts and the circles represent conceptual relations. The top concept [ARCH] is linked by the three conceptual relations (PART) to two [BRICK] concepts and a more general concept [PHYSOBJ]. Each [BRICK] is in a state (STAT) of [STAND]. The [PHYSOBJ] is in the state [LIE], supported (SUPP) by each [BRICK]. One [BRICK] has the other [BRICK] to its right (RGHT), and the two bricks do not abut (¬ABUT).

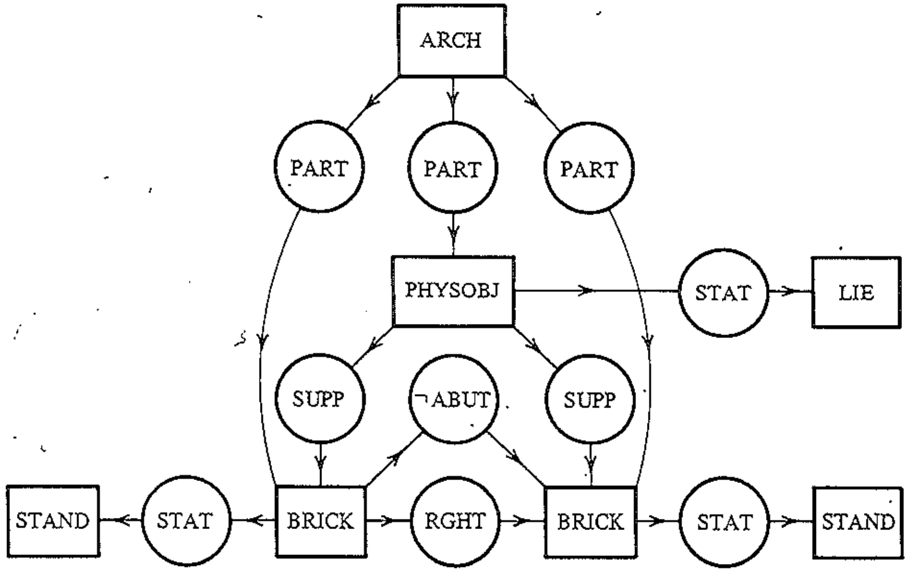

Fig. 3.2 Conceptual graph for an arch

In diagrams, a concept is drawn as a box, a conceptual relation as a circle, and an arc as an arrow that links a box to a circle. In linear text, the boxes may be abbreviated with square brackets, as in [ARCH], and the circles with rounded parentheses, as in (PART). To remember which way to draw the arrows on the arcs, read each subgraph of the form,

$$[\mathrm{CONCEPT}_1]\rightarrow(\mathrm{REL})\rightarrow[\mathrm{CONCEPT}_2]$$

as an English phrase, *the REL of a CONCEPT*$_1$ *is a CONCEPT*$_2$. If the label on the relation is not an English noun, this convention may lead to readings that are not grammatical as in *the RIGHT of a BRICK is a BRICK*. Even when the reading is ungrammatical, it still serves as a memory aid for drawing the arrows. These readings are helpful mnemonics, not linguistic rules for mapping the graphs into English. Section 5.4 presents grammar rules for generating natural language. Depending on context, a graph could be mapped into multiple sentences, a single sentence, a noun phrase, a verb phrase, or just a single word.

Conceptual relations may have any number of arcs, although most of the common ones are dyadic. Some, such as the *past tense marker* (PAST) or the *negation* (NEG), are monadic; others, like *between* (BETW), are triadic; and ones defined by the techniques of Section 3.6 may have an arbitrary number of arcs. Figure 3.3 shows a conceptual graph for the phrase *a space is between a brick and a brick*. For *n*-adic relations, the *n*th arc is drawn as an arrow pointing away from the circle, and all the other arcs are drawn as arrows pointing towards the circle. If the relation is monadic or dyadic, this convention is sufficient to distinguish the arcs. If $n \geq 3$, the arrows pointing towards the circle are numbered from 1 to n-1.

Conceptual graphs are finite, connected, bipartite graphs. They are *finite* because any graph in the human brain or computer storage can have only a finite number of concepts and conceptual relations. They are *connected* because two parts that were not connected would simply be called two conceptual graphs. They are *bipartite* because there are two different kinds of nodes—concepts and conceptual relations—and every arc links a node of one kind to a node of the other kind. For a survey of these and other terms from graph theory, see Appendix A.3.

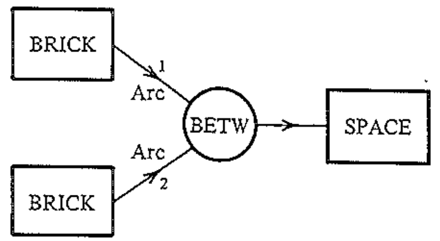

Fig. 3.3 Triadic relation "a space is between a brick and a brick"

3.1.2 Assumption. A *conceptual graph* is a finite, connected, bipartite graph.

- The two kinds of nodes of the bipartite graph are concepts and conceptual relations.
- Every conceptual relation has one or more *arcs*, each of which must be linked to some concept.
- If a relation has *n* arcs, it is said to be *n-adic*, and its arcs are labeled 1, 2, ..., *n*. The term *monadic* is synonymous with 1-adic, *dyadic* with 2-adic, and *triadic* with 3-adic.
- A single concept by itself may form a conceptual graph, but every arc of every conceptual relation must be linked to some concept.

Although diagrams help people to visualize relationships, the theory is independent of the way the graphs are drawn. Only three assumptions, which were analyzed in Section 2.3, are necessary to justify Assumption 3.1.2:

- Concepts are discrete units.
- Combinations of concepts are not diffuse mixtures, but ordered structures.
- Only discrete relationships are recorded in concepts. Continuous forms must be approximated by patterns of discrete units.

The notation of boxes and circles is a convenience, but it is not fundamental. What is fundamental is the mathematical basis, which is general enough to represent any set of relations between discrete things.

For concrete entities like cats and tomatoes, the brain has percepts for recognizing the entity and concepts for thinking about it. People can think about a restaurant as a place for eating food without imagining all the details of red checkered tablecloths and strolling musicians. For abstract types like JUSTICE and HEALTH, only imageless concepts, not percepts, are available. People differ widely in their use of concrete images and abstract concepts, but everybody (except perhaps those with serious brain injuries) is able use both.

3.1.3 Assumption. For every percept *p*, there is a concept *c*, called the *interpretation* of *p*. The percept *p* is called the *image* of *c*. Some concepts have no images.

- If a concept *c* has an image *p*, then *c* is called a *concrete concept*.
- If the concept *c* has no image, then *c* is called an *abstract concept*.
- The image of the interpretation of a percept *p* is identical to *p*.
- Entities recognized by the image of a concrete concept *c* are called *instances* of *c*.

Besides using conceptual graphs for interpreting sensory icons, the brain can also use them for generating or imagining new icons that were never before seen or heard.

A single percept or a structure of percepts can be *activated* to form an *imagined icon* that resembles a sensory icon. A conceptual graph serves as a plan for assembling percepts that form the internal image.

3.1.4 Assumption. Let u be a conceptual graph, whose concepts $c_1,...,c_n$ are all concrete. Then the graph u can serve as a pattern for a neural excitation t called an *imagined icon*. The icon t is identical to a sensory icon s with the following properties:

- The icon s may be matched by percepts $p_1,...,p_n$ where each p_i is the image of the concept c_i in the graph u.
- In matching the percepts $p_1,...,p_n$ to s, the assembler would construct a conceptual graph v identical to u.

The principle of assembling discrete patterns to form a continuous image is illustrated by the kits of plastic overlays used by police departments to form a picture of a crime suspect. The kits have a selection of plastic noses, chins, eyebrows, scars, hairlines, and so forth. The witness picks features from each category and overlays them to form a face. With Gillenson's WHATSISFACE system (1974) for automating the police kits, a witness sits in front of a display screen and answers questions like the following:

```
Is the top of the hairline
   (1) Evenly rounded or flattened
   (2) Parted on the left
   (3) Parted on the right
   (4) Fairly even but fairly pointy
   (5) Bunched up right in the middle
```

As the witness answers questions, selects sample features from a menu; and makes fine adjustments, the system assembles the features and displays a face that comes to resemble the original subject more and more closely. Figure 3.4 shows how well the system works: on the right are two photographs that users of the system were trying to duplicate, and on the left are the two matching faces constructed by the system. By combining discrete patterns, Gillenson's system can form a reasonable approximation to a continuous image. Besides assembling patterns, it also transforms and rotates the image to lengthen, shorten, or adjust the shape of the face and its features. The human brain also performs such transformations in assembling percepts and matching them to icons, but long-term memory quickly loses continuous information about sizes and rotations.

Many pattern recognition and generation systems analyze continuous images in terms of discrete units. By analogy with grammars in linguistics, which define permissible combinations of words, such formalisms are commonly called *picture*

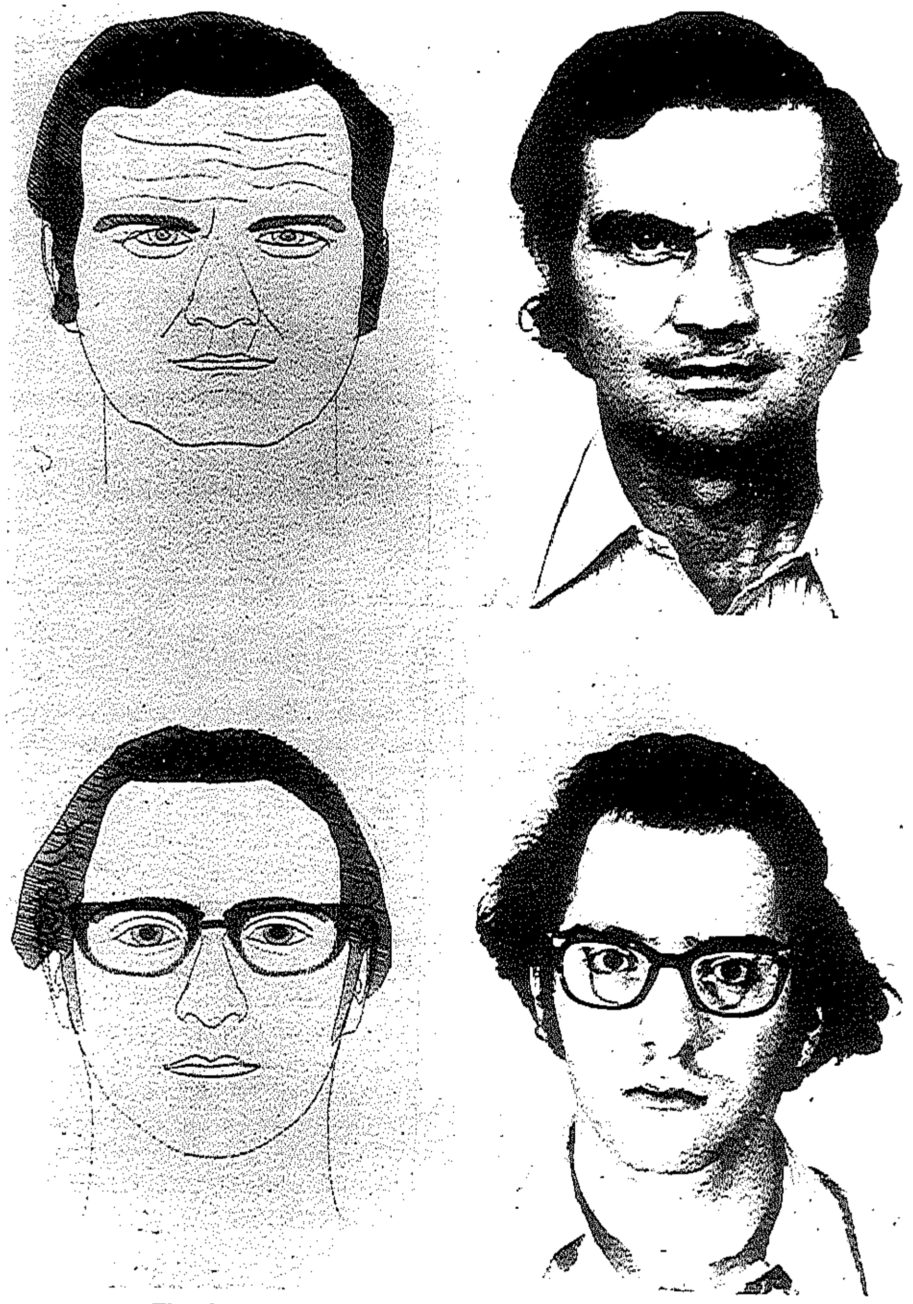

Fig. 3.4 Two faces constructed by assembling patterns

grammars or *shape grammars*. Ledley (1964) wrote a grammar for chromosomes that was based on combinations of curves; he then developed a parsing program that could analyze chromosomes in a photograph and classify them according to the grammar. Even movements of the human body can be described in discrete units. Labanotation (Hutchinson 1970) was originally developed as a language for describing dance movements, but Badler and Smoliar (1979) wrote programs to translate Labanotation into animated images on a computer display. Although these notations are different from conceptual graphs, they, too, illustrate the principle of generating continuous images from discrete plans.

3.2 SEMANTIC NETWORK

Although the concept types CAT and TOMATO map directly to percepts, other types like PRICE, FUNCTION, and JUSTICE have no sensory correlates. Abstract concepts acquire their meaning not through direct associations with percepts, but through a vast network of relationships that ultimately links them to concrete concepts. In philosophy, White (1975) considered the meaning of a concept to be its position in that network:

> To discover the logical relations of a concept is to discover the nature of that concept. For concepts are, in this respect, like points; they have no quality except position. Just as the identity of a point is given by its coordinates, that is, its position relative to other points and ultimately to a set of axes, so the identity of a concept is given by its position relative to other concepts and ultimately to the kind of material to which it is ostensively applicable.... A concept is that which is logically related to others just as a point is that which is spatially related to others.

The collection of all the relationships that concepts have to other concepts, to percepts, to procedures, and to motor mechanisms is called the *semantic network*.

Some concepts in the semantic network are firmly anchored to percepts, but others apply to the ethereal realms of religion, economics, and theoretical physics. Yet concrete concepts are not necessarily more "meaningful" than abstract ones. A physical description of eight hours in a person's life may be an incomprehensible mass of detail, but the detail might be explained by the sentence *Leon had a hectic day at the stock exchange because he is a specialist in XYZ stock and rumors say that another company is trying to buy out XYZ.* Anyone who fails to appreciate the abstraction of this sentence should try defining the types RUMOR, STOCK, and BUY-OUT in purely sensory terms. BUY is a TRANSACTION in which the buyer exchanges MONEY for the OWNERSHIP of some ENTITY. But what are the sensory correlates of OWNERSHIP and MONEY? What distinguishes RUMOR from FACT, FACT from UTTERANCE, or UTTERANCE from SOUND?

A conceptual graph has no meaning in isolation. Only through the semantic network are its concepts and relations linked to context, language, emotion, and perception. Figure 3.5 shows a conceptual graph for *a cat sitting on a mat*. Dotted lines link the nodes of the graph to other parts of the semantic network:

- Concrete concepts are associated with percepts for experiencing the world and motor mechanisms for acting upon it.
- Some concepts are associated with the words and grammar rules of a language.
- A hierarchy of concept types defines the relationships between concepts at different levels of generality.
- Formation rules determine how each type of concept may be linked to conceptual relations.
- Each conceptual graph is linked to some context or episode to which it is relevant.
- Each episode may also have emotional associations, which indirectly confer emotional overtones on the types of concepts involved.

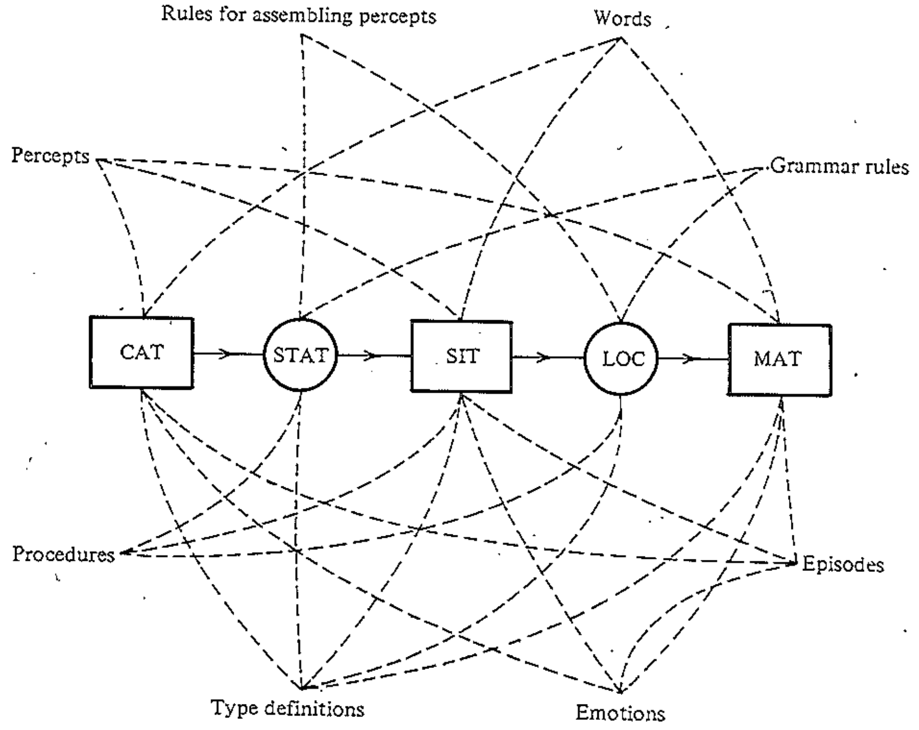

Fig. 3.5 A conceptual graph linked to the semantic network

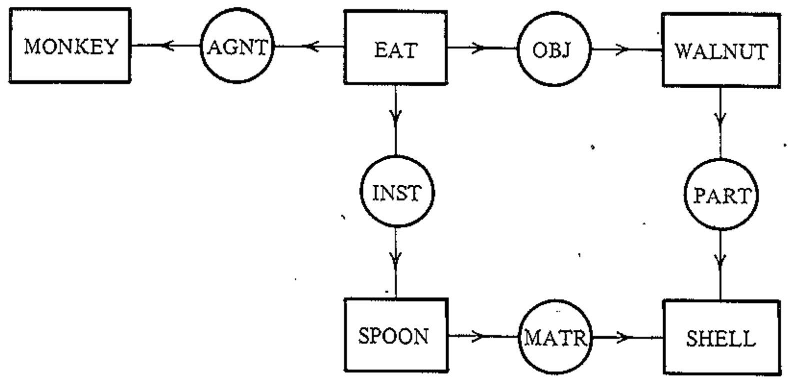

Fig. 3.6 Conceptual graph with a cycle

Some authors use the term *semantic network* in a way that is almost synonymous with *conceptual graph.* In this book, however, they are distinguished. Each conceptual graph asserts a single proposition. The semantic network is much larger. It includes a defining node for each type of concept, subtype links between the defining nodes, and links to perceptual and motor mechanisms. Some aspects of the semantic network, such as the type hierarchy, are treated in great detail. Other aspects, such as emotions, are not discussed in detail—not because they are unimportant, but because no one knows how to treat them formally.

The box and circle notation, called the *display form* for conceptual graphs, cannot be typed easily at a computer terminal. For the cat sitting on a mat in Fig. 3.5, the graph is already linear:

```
[CAT]→(STAT)→[SIT]→(LOC)→[MAT].
```

If any concept or relation is linked to more than two arcs, the graph forms a tree. If there are cycles in the graph, *variables* are needed to show cross references. As an example of a conceptual graph that cannot be represented in either a line or a tree, Fig. 3.6 may be read, *a monkey eating a walnut with a spoon made out of the walnut's shell.* (See Appendix B.3 for the conceptual relations used in these examples.) In the *linear form* for conceptual graphs, some concept or relation must be the *head.* Usually, the result is simplest if the concept with the most arcs linked to it is chosen as head:

```
[EAT]-
   (AGNT)→[MONKEY]
   (OBJ)→[WALNUT:*x]
   (INST)→[SPOON]→(MATR)→[SHELL]←(PART)←[WALNUT:*x].
```

The symbol *x is called a *variable.* It represents an unspecified individual of type WALNUT—which one is not known, but it must be the same one in both occurrences. The hyphen "-" after the concept [EAT] shows that the relations connected to [EAT] are listed on subsequent lines; the period "." terminates the entire graph. Any other concept or relation could have been chosen as head, but the linear form might be more complicated:

```
[SPOON]-
   (INST)←[EAT]-
            (OBJ)→[WALNUT]→(PART)→[SHELL:*y]
            (AGNT)→[MONKEY],
   (MATR)→[SHELL:*y].
```

Both of these linear forms are exactly equivalent to Fig. 3.6 and to each other. The differences between them result from arbitrary choices: which node to pick as the head, which relation to list first in the linear form, and which node to mark with a variable when breaking cycles. In this second form, both [SPOON] and [EAT] are followed by hyphens to show that their relations are listed on subsequent lines. A comma terminates the hyphen after [EAT] so that (MATR) is linked by the previous hyphen on [SPOON]. The hyphen and comma form a *bracketing pair* that is necessary for linearizing a tree. Indentation makes the bracketing more readable. For the detailed syntax of the linear form, see Appendix A.6.

Normally, the entire semantic network is not drawn explicitly because it is too large and unwieldy. Instead, each concept box contains a label that shows the type, and two boxes with the same *type label* represent concepts of the same type. The distinction between type labels and concepts follows the distinction between *types* and *tokens* drawn by Peirce (1906): the word *cat* is a type, and every utterance of *cat* is a new token. Similarly, each occurrence of a concept is a separate token, but the tokens are classified by a set T of basic types.

3.2.1 Assumption. The function *type* maps concepts into a set T, whose elements are called *type labels.* Concepts c and d are of the *same type* if $type(c)=type(d)$.

Some graph notations leave the concept boxes empty, and represent types by special *IS-A links* to a defining node for each type. For many purposes, the two conventions have the same effect, but writing type labels in the boxes instead of IS-A links makes conceptual graphs easier to read. Besides readability, there are theoretical reasons for not drawing IS-A links as circles: conceptual relations show the role that each concept plays; the type hierarchy is a *higher-order relation* not between individual concepts, but between types of individuals. In this book, type labels are written as character strings: RED and PHYSOBJ are type labels for concepts of redness and physical object. A concept of type RED is drawn as a box in a diagram or with square brackets [RED] in the text. For readability, English words in upper case are used for most type labels, but any set of unique identifiers such as numbers or Chinese characters would work just as well.

Whether two concept types are the same depends on their links to the semantic network rather than their external instances. This distinction is important because some types may not have corresponding sets, and some sets may not have types. The set of type UNICORN, for example, is empty. Furthermore, a set of unrelated things like an elephant, a paper clip, and the moon does not correspond to a type. No single percept could recognize all three of them without also recognizing every other rounded object in the world. Instances of the same type have a natural affinity for each other, at least in the eye of some beholder. An arbitrary set, however, may not have such an affinity. For a *natural type* like ROSE or CAT, the instances are related by their nature, not by an arbitrary selection. All the things in the real world that are instances of a type constitute the *denotation* of that type.

3.2.2 Definition. Let t be a type label. The *denotation* of type t, written δt, is the set of all entities that are instances of any concept of type t.

Since no two stimuli are ever identical in all respects, every percept must match a range of sensory icons. A percept that matches a broad range of icons is more general than one that matches only a subrange. The image of type RED is a percept that matches an infinite variety of hues, including those matched by percepts for STRAWBERRY-RED, FIRE-ENGINE-RED, CRIMSON, and SCARLET. Since RED is the label of a more general concept than CRIMSON, the type CRIMSON is called a *subtype* of RED. The denotation of CRIMSON is a subset of the denotation of RED: δCRIMSON is contained in δRED. The symbol $\le$ represents subtypes: CRIMSON$\le$RED, and RED$\ge$CRIMSON. Every type is a subtype of itself: RED$\le$RED. The symbols $<$ and $>$ exclude the possibility that two labels are equal: CRIMSON$<$RED, but not RED$<$RED. But not all types of concepts are comparable: neither RED nor JUSTICE is a subtype of the other. See Appendix A.4 for a definition of partial ordering.

3.2.3 Assumption. The *type hierarchy* is a partial ordering defined over the set of type labels. The symbol $\le$ designates the ordering. Let s, t, and u be type labels:

- If $s \le t$, then s is called a *subtype* of t; and t is called a *supertype* of s, written $t \ge s$.
- If $s \le t$ and $s \ne t$, then s is called a *proper subtype* of t, written $s<t$; and t is called a *proper supertype* of s, written $t>s$.
- If s is a subtype of t and a subtype of u ($s \le t$ and $s \le u$), then s is called a *common subtype* of t and u.
- If s is a supertype of t and a supertype of u ($s \ge t$ and $s \ge u$), then s is called a *common supertype* of t and u.

In AI, the type hierarchy supports the inheritance of properties from supertypes to subtypes of concepts. Aristotle first introduced type hierarchies with his theory of categories and syllogisms. He had ten primitive types, a method for defining new types by *genus* and *differentiae*, and the use of syllogisms for analyzing the inheritance of properties. Masterman (1961) introduced the term *semantic net* for the type hierarchy and defined an algorithm that let subtypes inherit the properties or *semantic shells* of supertypes.

The best way to study type hierarchies is to analyze the structure of dictionary definitions, preferably by computer. Amsler (1980) found a rich hierarchy in his analysis of the *Merriam-Webster Pocket Dictionary*. The hierarchy tended to be bushy, with each node having many descendants, but it did not grow very deep. The concept type VEHICLE, for example, had 165 subtypes, but the hierarchy extended for only three levels. At the first level, the immediate subtypes of VEHICLE included AMBULANCE, AUTOMOBILE, BICYCLE, BUCKBOARD, BUS, CARRIAGE, CART, etc. The next level beneath AUTOMOBILE included the subtypes COACH, CONVERTIBLE, COUPE, HOT-ROD, JALOPY, SEDAN, etc. The third level beneath SEDAN included BROUGHAM, LIMOUSINE, and SALOON. Actions, states, and properties can also be grouped in hierarchies. In analyzing verbs, Chodorow (1981) also found bushy, but shallow hierarchies. The concept type COMPLAIN, for example, has subtypes BELLYACHE, BITCH, CRAB, GRIPE, INVEIGH, SQUAWK, and WHIMPER, none of which have any further subtypes.

Corresponding to the type hierarchy for concepts is an *approximation hierarchy* for percepts. A percept for a general type RED makes an approximate match to many different icons. A percept for the subtype CRIMSON matches fewer icons, but it matches them more exactly.

3.2.4 Assumption. The *approximation hierarchy* is a partial ordering of percepts induced by the partial ordering of concept types. If the percept p is the image of a concept of type s and q is the image of a concept of type t where $s \le t$, then define $p \le q$. The following conditions hold:

- Any entity recognized by p is also recognized by q.
- Hence, the denotation of s is a subset of the denotation of t: $\delta s \subseteq \delta t$.
- If an icon i is matched by both percepts p and q, the percept p forms a more exact match to i than the percept q.

The types CAT and DOG have many common supertypes, including ANIMAL, VERTEBRATE, MAMMAL, and CARNIVORE. The *minimal common supertype* of CAT and DOG is CARNIVORE, which is a subtype of all the other supertypes. The concept types FELINE and WILD-ANIMAL have common subtypes

JAGUAR, LION, and TIGER; but none of them is a *maximal common subtype*. The type hierarchy could be refined, however, by adding the type WILD-FELINE, which would be a maximal common subtype of FELINE and WILD-ANIMAL and a minimal common supertype of LION, TIGER, and JAGUAR. When all the intermediate types like WILD-FELINE are introduced, the type hierarchy becomes a *type lattice*. Both Leibniz's Universal Characteristic and Masterman's semantic nets were lattices. See Appendix A.4 for a definition of *lattice*.

3.2.5 Assumption. The type hierarchy forms a lattice, called the *type lattice*:

- Any pair of type labels s and t has a *minimal common supertype*, written $s \cup t$. For any type label u, if $u \geq s$ and $u \geq t$, then $u \geq s \cup t$.
- Any pair of type labels s and t has a *maximal common subtype*, written $s \cap t$. For any type label u, if $u \leq s$ and $u \leq t$, then $u \leq s \cap t$.
- There are two primitive type labels: the *universal type* $\top$ and the *absurd type* $\bot$. For any type label t, $\bot \leq t \leq \top$.

As supertypes of CAT and DOG, the previous discussion did not mention the concept type PET. There is a reason for that omission: the types CAT, DOG, MAMMAL, and ANIMAL are *natural types* that relate to the essence of the entities, but types like PET, PEDESTRIAN, and SPOUSE are *role types* that depend on an accidental relationship to some other entity. Any animal may be a pet if it plays the proper role: a person could have a pet mouse, pet alligator, or pet crab. Even the chimpanzee Washoe had a pet cat. Natural types and role types both occur in the same type lattice. The maximal common subtype of CAT and PET is PET-CAT; the minimal common supertype of PET-CAT and PET-DOG is PET-CARNIVORE.

A lattice must have minimal common supertypes and maximal common subtypes. Yet for most pairs of types, such as CAT and APPLE, SQUARE and CIRCLE, or JUSTICE and DOG, there are no common subtypes. To make the type hierarchy into a lattice, two special type labels must be introduced at its top and bottom: the universal type $\top$ that is a supertype of all other types, and the absurd type $\bot$ that is a subtype of all other types. Everything that exists is an instance of the universal type $\top$. No actual entity could ever be an instance of type $\bot$, since it would have to be a dog, a cat, a circle, a square, an apple, and justice as well as an event, a color, and an emotion, all at the same time.

Many people confuse types and sets. Yet there is a fundamental difference. Statements about types are *analytic*; they must be true by intension. Statements about sets are *synthetic*; they are verified by observing the extensions. To say that the intersection of the set δCAT with the set δDOG is empty simply means that at the moment no individual happens to be both a dog and a cat. A biologist might examine litters of puppies and kittens looking for an exception. But to say that

DOG $\cap$ CAT $= \bot$ means that it is logically impossible for an entity to be both a dog and a cat. Any mutant that might arise could not falsify the statement; it would just force the biologist to invent a new type.

Although the operators $\cup$ and $\cap$ are used for both types and sets, there are important differences in the way they behave on type labels and denotations. For extensions, the union δCAT $\cup$ δDOG is the set of all cats and dogs in the world and nothing else. But for the intensional type labels, CAT $\cup$ DOG is their minimal common supertype CARNIVORE, which also has subtypes BEAR, WEASEL, SKUNK, etc. Since the union of any set S with the empty set leaves S unchanged, the union δCAT $\cup$ δUNICORN is just δCAT. Yet the denotation of UNICORN is empty only in the present world. In some possible world, unicorns might exist. Even in this world, a molecular geneticist may someday find a way of creating one. To allow for such possibilities, CAT $\cup$ UNICORN must be MAMMAL. Even more dramatically, JUSTICE $\cup$ UNICORN is the universal type $\top$, which includes everything. The type lattice represents categories of thought, and the lattice of sets and subsets represents collections of existing things. The two lattices are not isomorphic, and the denotation operator that maps one into the other is neither one-to-one nor onto.

3.2.6 Theorem. Let s and t be any type labels. Then $\delta(s \cup t)$ is a superset of $\delta s \cup \delta t$, and $\delta(s \cap t)$ is a subset of $\delta s \cap \delta t$.

Proof. By definition, both s and t are subtypes of $s \cup t$. Hence by Assumption 3.2.4, any element of δs or of δt must also be an element of $\delta(s \cup t)$. Therefore, $(\delta s \cup \delta t) \subseteq \delta(s \cup t)$. Since $s \cap t$ is a subtype of both s and t, any element of $\delta(s \cap t)$ must be an element of δs and of δt. Therefore, $\delta(s \cap t) \subseteq (\delta s \cap \delta t)$.

Conceptual relations are classified by types in the same way that concepts are classified. A hierarchy is also defined over type labels of conceptual relations. A general relation type LOC for location may have subtypes that specify more details about the location, such as IN, ABOV, or UNDR.

3.2.7 Assumption. The function *type* may be extended to map conceptual relations to type labels.

- The relations r and s are said to be of the *same type* if $type(r)=type(s)$.
- If r and s are of the same type, they must have exactly the same number of arcs.
- For any concept c and conceptual relation r, $type(c) \neq type(r)$.
- The partial ordering of type labels also extends to type labels of conceptual relations, but the type labels of concepts have no common supertype with the type labels of conceptual relations.

3.3 INDIVIDUALS AND NAMES

George Washington and *Grand Central Terminal* are proper names, but *building, lawyer, cat,* and *pedestrian* are common nouns. English and other natural languages distinguish the two kinds of words. A proper name is an arbitrary label that designates a particular person, place, thing, or group. A common noun is a generic term that specifies some attribute of the entities it refers to, but it applies equally well to any entity that has that attribute. Even when a proper name describes some attribute, it remains an arbitrary label: Mont Blanc may be a white mountain, but no other white mountain is named Mont Blanc. In general, common nouns correspond to type labels, and names designate particular individuals. Confusing the two categories of words can easily lead to absurd fallacies:

```
If Clyde is an elephant,              If Clyde is an elephant,
and an elephant is an animal,         and elephant is a species,
then Clyde is an animal.              then Clyde is a species.
```

The statement *Clyde is an elephant* says that a particular individual named Clyde is of type ELEPHANT. The statement *An elephant is an animal* says that any individual of type ELEPHANT is also of type ANIMAL. The first syllogism is therefore valid. But the statement *Elephant is a species* is not directly about individuals. It states that the word *elephant* is the name of a particular species of animal. In that syllogism, the middle term *elephant* is used in two different senses—one of the classic forms of fallacy.

The concepts defined so far are *generic concepts*: they are like variables that represent an unspecified individual of a given type. To distinguish types from individuals, some new features must be added. For database systems, Hall, Owlett, and Todd (1976) introduced unique identifiers or *surrogates* to identify particular individuals. For relational database theory, Codd (1979) adopted surrogates as internal representatives of external entities.

Codd distinguished the *perceived world* from the real world. Surrogates in a database do not directly reflect entities in the real world, but in some system analyst's perception of what is significant and what should be stored. Besides surrogates for specific individuals, a database may also contain *null values*, which are place holders for individuals whose identities are unknown. A database of family relationships, for example, would either have null values for the parents of some people or it would have to contain everybody's genealogy back to Adam and Eve—even then it would have to have null values for the parents of those two.

In conceptual graphs, surrogates are represented by *individual markers*, which are serial numbers like #80972, and null values are represented by asterisks. The concept box is divided into two fields separated by a colon, as in [PERSON:#80972]. The field to the left of the colon contains the type label PERSON. The field to the right of the colon contains the *referent* #80972, which designates a particular person.

If the referent is just an asterisk, as in [PERSON:*], the concept is called a *generic concept*, which may be read *a person* or *some person*. In diagrams, the asterisk is optional. A box with just a type label like [PERSON] is equivalent to [PERSON:*].

3.3.1 Assumption. There is a set I={#1, #2, #3,...} whose elements are called *individual markers*. The function *referent* may be applied to any concept c:

- *referent*(c) is either an individual marker in I or the *generic marker* *.
- When *referent*(c) is in I, then c is said to be an *individual concept*.
- When *referent*(c) is *, then c is said to be a *generic concept*.

An individual marker is a surrogate for some individual in the real world, a perceived world, or a hypothetical world. Any type of concept may have an individual marker: the concept [CAT] or [CAT:*] refers to an unspecified cat, but [CAT:#98077] refers to a particular individual with serial number #98077; the concept [JUMP] refers to an unspecified act of jumping, but [JUMP:#882635] refers to a particular instance of jumping; [HAPPY] refers to some happiness, but [HAPPY:#77289] refers to a particular instance of happiness. Individual concepts can represent *mass nouns* like water or butter and *count nouns* like boy or pencil. The concept [WATER:#2219] would represent a particular mass of water. Having individual markers on mass concepts is necessary to distinguish a new batch of water flowing through a fountain from an old batch recycled.

In a computer simulation, each individual marker is represented by a unique number or symbol. In the human brain, a marker could be represented by an association to the episode or context to which the conceptual graph was originally linked. In that case, a marker is like a special time stamp that specifies the time and place when the concept was recorded. Following is the psychological evidence for such markers:

- All human languages distinguish proper names from common nouns.
- In computer systems, the simplest way of identifying entities is by assigning each a unique marker, such as a time stamp or serial number.
- Macnamara (1982) found that the distinction between proper names and common nouns is among the first ones that children learn, some before the age of 17 months.

Since children learn names so quickly, names must be logically close to the structures that the brain uses for identifying individuals. A name is a perceptible sound or symbol associated with the internal, neural marker. In linguistics, Chomsky (1965) introduced unique indices to specify which noun phrases refer to the same individuals. In psychology, Norman, Rumelhart, et al. (1975) used similar markers for individual concepts.

Individual concepts correspond to constants in logic and programming languages, and generic concepts correspond to variables. In fact, variables like *x or *y in the linear notation are simply the generic marker *, followed by an identifier for indicating cross references. The following two graphs are exactly equivalent:

```
[CAT:*x]→(STAT)→[SIT:*z]→(LOC)→[MAT:*y].
[CAT]→(STAT)→[SIT]→(LOC)→[MAT].
```

Both of these graphs make the same assertion: they say that there exists some cat x, some mat y, and some instance of sitting z by cat x on mat y. If the concept [CAT] had the referent #98077, it would still assert existence, but it would be a definite reference *the cat* instead of an indefinite *a cat*. If the concept [SIT] had an individual marker, the graph could be read *the sitting by a cat on a mat*. Every generic concept is bound by an implicit *existential quantifier*. The next assumption introduces the *formula operator*, represented by the Greek letter ϕ, which translates a conceptual graph into a logical formula.

3.3.2 Assumption. The operator ϕ maps conceptual graphs into formulas in the first-order predicate calculus. If u is any conceptual graph, then ϕu is a formula determined by the following construction:

- If u contains k generic concepts, assign a distinct variable symbol $x_1,x_2,\ldots,x_k$ to each one.
- For each concept c of u, let *identifier*(c) be the variable assigned to c if c is generic or *referent*(c) if c is individual.
- Represent each concept c as a monadic predicate whose name is the same as *type*(c) and whose argument is *identifier*(c).
- Represent each n-adic conceptual relation r of u as an n-adic predicate whose name is the same as *type*(r). For each i from 1 to n, let the ith argument of the predicate be the identifier of the concept linked to the ith arc of r.
- Then ϕu has a *quantifier prefix* $\exists x_1 \exists x_2 \ldots \exists x_k$ and a *body* consisting of the conjunction of all the predicates for the concepts and conceptual relations of u.

Every conceptual graph makes an assertion. The operator ϕ maps that assertion into a logical formula. In the graphs of this chapter, the only logical operators are the conjunction $\wedge$ and the existential quantifier $\exists$. Chapter 4 introduces negation and then defines other operators in terms of $\sim$ and $\wedge$ together with $\exists$. For a review of standard logic, see Appendix A.4. As an example of Assumption 3.3.2, let the graph u be,

```
[CAT:#98077]→(STAT)→[SIT]→(LOC)→[MAT].
```

Since this graph has two generic concepts, assign one variable x to [SIT] and another variable y to [MAT]. Then the three concepts map to the monadic predicates CAT(#98077), SIT(x), and MAT(y). The two conceptual relations map to STAT(#98077,x) and LOC(x,y). Then ϕu is

$$\exists x \exists y(\mathrm{CAT}(\#98077) \wedge \mathrm{STAT}(\#98077,x) \wedge \mathrm{SIT}(x) \wedge \mathrm{LOC}(x,y) \wedge \mathrm{MAT}(y))$$

When a graph u is mapped into a formula ϕu, the individual concepts of u are mapped to constants, and the generic concepts are mapped to variables. Since a single concept by itself forms a conceptual graph, the operator ϕ maps the concept [PERSON] to the formula $\exists x$PERSON(x). Conceptual graphs are usually more concise than logical formulas because arcs on the graphs show the connections more directly than variable symbols.

An individual marker on a concept must conform to its type label. Any operation that might change the type label must check whether the new label is appropriate for the old individual. If the individual #98077 happened to be a dog, the concept [ANIMAL:#98077] could not be restricted to [CAT:#98077]. The *conformity relation*, denoted by the symbol ::, makes the test: if #98077 is a cat, then CAT::#98077 is true; otherwise, it is false. Note the difference between the single colon and the double colon. In the concept [CAT:#98077], the single colon is just punctuation that separates the type field from the referent field. In the conformity relation CAT::#98077, the double colon is an operator that makes a test.

Suppose that Snoopy is a beagle identified by the marker #2883. Then the relation BEAGLE::#2883 states that the individual #2883 conforms to type BEAGLE and therefore that the concept [BEAGLE:#2883] is well formed. Because of the type hierarchy, the following relationships must also be true: DOG::#2883, MAMMAL::#2883, ANIMAL::#2883, PHYSOBJ::#2883, and ENTITY::#2883. If Snoopy is a beagle and Snoopy is a pet, then Snoopy must also be a pet beagle. BEAGLE::#2883 and PET::#2883 imply PET-BEAGLE::#2883.

3.3.3 Assumption. The *conformity relation* :: relates type labels to individual markers: if t::i is true, then i is said to *conform* to type t. The conformity relation obeys the following conditions:

- The referent of a concept must conform to its type label: if c is a concept, *type*(c)::*referent*(c).
- If an individual marker conforms to type s, it must also conform to all supertypes of s: if $s \leq t$ and s::i, then t::i.
- If an individual marker conforms to types s and t, it must also conform to their maximal common subtype: if s::i and t::i, then $(s \cap t)$::i.
- Every individual marker conforms to the universal type $\top$; no individual marker conforms to the absurd type $\bot$: for all i in I, $\top$::i, but not $\bot$::i.
- The generic marker * conforms to all type labels: for all type labels t, t::*.

Once the theory can identify individuals, it has a basis for defining names of individuals. The concept ["Judy"] is a concept of the name *Judy*, as opposed to [PERSON:Judy], which is a concept of the person Judy. The concept ["Judy"] is generic because it is possible to talk about the utterance #423781 of the name *Judy*, which is represented by the concept ["Judy":#423781]. The conceptual relation (NAME) links a concept to a word that names an individual. Only entities can have names. Individuals of other types, such as ACT, cannot be named.

3.3.4 Assumption. NAME is a type label for a dyadic conceptual relation, and ENTITY and WORD are type labels for concepts. Let a and b be any concepts linked to arcs #1 and #2 of a conceptual relation of type NAME: a→(NAME)→b. Then the following conditions must hold:

- *type*(a) is a subtype of ENTITY: *type*(a)≤ENTITY.
- *type*(b) is a proper subtype of WORD: *type*(b)<WORD.

The word *type*(b) is called a *name* of *referent*(a).

The following graph represents a PERSON identified by individual marker #3074, who has a name *Judy*:

```
[PERSON:#3074]→(NAME)→["Judy"].
```

Because names occur so frequently, this graph can be abbreviated by *name contraction* to form just the single concept [PERSON:Judy#3074]. This form corresponds to the English phrase *the person Judy*, which is a contraction of *the person named Judy*. The following graph,

```
[PERSON]→(NAME)→["Judy"],
```

may be contracted to the single concept [PERSON:Judy]. This concept represents the indefinite form *a person named Judy*, but [PERSON:Judy#3074] represents *the person named Judy*. If there is only one person named Judy in the context, the distinction may not matter. If the name *Snoopy* uniquely identifies the individual #2883, the name could also be used in the conformity relation BEAGLE::Snoopy instead of BEAGLE::#2883.

A person may have multiple names or aliases. The following graph shows a person #1196 who has the names Cicero and Tully:

```
["Cicero"]←(NAME)←[PERSON:#1196]→(NAME)→["Tully"].
```

By name contraction, either name could be contracted into the referent field of the concept [PERSON], and the individual marker #1196 could be dropped:

```
[PERSON:Cicero]→(NAME)→["Tully"].
[PERSON:Tully]→(NAME)→["Cicero"].
```

The first graph may be read *Cicero is named Tully*; and the second, *Tully is named Cicero*. The NAME relation applies equally well to names of entities in the real world, other possible worlds, or Platonic realms of ideas. The statement *The present king of France is named Louis XXIV* is mapped to a normal conceptual graph, even though some of the concepts in it may not represent anything in the real world.

Mathematical entities like numbers, sets, and functions can also be represented and named. The English word *four*, the French word *quatre*, the Arabic numeral 4, the Roman numeral IV, and the binary numeral 100 are all names for the same number. The next graph shows the number with two of its names:

```
["IV"]←(NAME)←[NUMBER:#27018]→(NAME)→["4"].
```

The individual marker #27018 distinguishes a concept of a unique number, which has several aliases. This graph would be contracted to [NUMBER:4] in normal computation. The example *Elephant is a species*, which appeared earlier in this section, could be represented by the graph,

```
["elephant"]←(NAME)←[SPECIES]→(MEMB)→[ELEPHANT:{*}].
```

This graph may be read *The word "elephant" is a name of a species whose members are a set of elephants*. The symbol {*}, which is defined in Section 3.7, represents a *generic set* of entities that conform to the type label of the concept.

Measures can be treated in the same way as names: *10 inches* and *25.4 centimeters* are alternative names for the same quantity of linear measure. Like numerals, units of measure belong to a system with rules for converting one name into another. Yet they are still names. *Kilogram* is the name given to the measure of a mass in Sèvres, France; other masses may also have that same measure. In the fully expanded form, conceptual graphs can distinguish a bar, the length of the bar, a measure of that length, and a name for that measure:

```
[BAR]→(CHRC)→[LENGTH]→(MEAS)→[MEASURE]→(NAME)→["25.4 cm"].
```

This graph may be read *a bar with a length with a measure named 25.4 cm*. By name contraction, it may be simplified to

```
[BAR]→(CHRC)→[LENGTH]→(MEAS)→[MEASURE: 25.4 cm].
```

This contracted graph may be read *a bar with a length with a measure of 25.4 cm*. Since units of measure occur so frequently, they may be abbreviated further by *measure contraction*:

```
[BAR]→(CHRC)→[LENGTH: @ 25.4 cm],
```

which may be read *a bar of length 25.4 cm*. The symbol @ shows that the following string is not a name, but a measure. If [LENGTH] originally had an individual marker #7286, then the contracted form would be [LENGTH:#7286@25.4cm].

The contractions for measures are similar to the abbreviations in English and other natural languages. A failure to distinguish the different forms of abbreviation can lead to confusions and paradoxes. Montague (1974), for example, noted a paradox in the sentence *The temperature is ninety but it is rising.* If the copula *is* were interpreted as a sign of equality, this sentence would lead to the absurd conclusion that ninety is rising. Montague resolved the paradox by distinguishing an individual concept from its measure at some instant of time. Although Montague's notation is very different, a similar analysis could be expressed in a conceptual graph:

```
[TEMPERATURE:#55361]→(MEAS)→[MEASURE]→(NAME)→["90°"].
```

This graph says that the instance of temperature designated by #55361 has a measure named 90°. If the temperature #55361 happened to rise, it would have a different measure, which would have a different name. But the measure named 90° would remain the same.

Formally, the referent of a concept is either * for generic concepts or # followed by an integer for individual concepts. The other notations written in the referent field of a concept node are informal abbreviations that can always be converted to the formal notation:

- In the linear form, the * for generic concepts may be followed by a variable name, such as *x or *abc.
- The marker @ shows that the following string is a measure of an entity and not a name or individual marker that designates the entity.
- A character string not preceded by a symbol *, #, or @ represents a name of an individual that must be unique within the current context.
- To represent the definite article, as in *the cat*, the integer following # may be dropped, as in [CAT:#] instead of [CAT:#98077]. This abbreviation is used when there is just a single cat in the context.

More complex referents are introduced in Section 3.7, including sets and nested conceptual graphs.

3.4 CANONICAL GRAPHS

A conceptual graph is a combination of concept nodes and relation nodes where every arc of every conceptual relation is linked to a concept. But not all such combinations make sense. Some of them include absurd combinations like the following:

```
[SLEEP]→(AGNT)→[IDEA]→(COLR)→[GREEN].
```

This is an odd, unusual, or perhaps meaningless graph that may be read *Some act of sleeping has an agent, which is an idea, which has a color, green.* This odd combination could have arisen from Chomsky's famous sentence *Colorless green ideas sleep furiously* (1957). To rule out such sentences, Katz and Fodor (1963) developed a theory of semantics that imposes *selectional constraints* on permissible combinations of words.

To distinguish the meaningful graphs that represent real or possible situations in the external world, certain graphs are declared to be *canonical.* Through experience, each person develops a world view represented in canonical graphs. One source of the graphs is observation: the assembler may combine certain concepts in perception. Since that combination is true of a real situation, it must be canonical. Another source is the derivation of new canonical graphs from other canonical graphs by *formation rules.* The third source is called *insight* or creativity: a person may feel that existing percepts, concepts, and relations do not adequately describe a situation and may invent a radically new configuration that better describes it. Insight leads to new canonical graphs that replace or extend the older ones.

3.4.1 Assumption. Certain conceptual graphs are *canonical.* New graphs may become canonical or be *canonized* by any of the following three processes:

- *Perception.* Any conceptual graph constructed by the assembler in matching a sensory icon is canonical.
- *Formation rules.* New canonical graphs may be derived from other canonical graphs by the rules *copy*, *restrict*, *join*, and *simplify.*
- *Insight.* Arbitrary conceptual graphs may be assumed as canonical.

Before graphs can be derived by formation rules, a starting set must be introduced by insight. A child invents graphs that provide the simplest interpretation of experience. When discrepancies between new experiences and the old graphs become too great, the child invents a newer, more adequate set of graphs. In a knowledge-based system, insight corresponds to the introduction of new graphs by a knowledge engineer. In experimental systems, insight is simulated by learning programs that search for graphs that encode information more efficiently.

The formation rules are a generative grammar for conceptual structures just as Chomsky's production rules are a generative grammar for syntactic structures. All deductions and computations on conceptual graphs involve some combination of them. There are several arguments for this choice of rules:

- *Common practice.* Similar operations have been useful in many AI programs.
- *Modularity.* They allow new graphs to be derived as combinations of standard modules or templates.
- *Selectional constraints.* They enforce constraints that prevent nonsensical combinations from being derived.
- *Theoretical elegance.* They simplify the definitions and proofs of the theory.

By the *copy* rule, an exact copy of a canonical graph is also canonical. The *restrict* rule replaces the type label of a concept with the label of a subtype, as in deriving [GIRL] from [PERSON]. It may also convert a generic concept like

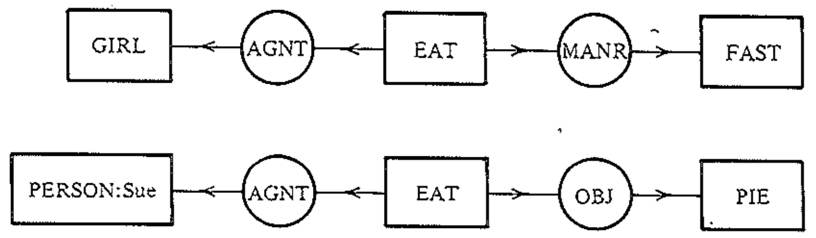

Fig. 3.7 Two canonical graphs

[DOG] to an individual concept [DOG:Snoopy]. For individual concepts, the conformity relation must be checked to make sure that the referent conforms to the new type label. A restriction from [DOG:Snoopy] to [BEAGLE:Snoopy] is permissible, but a restriction to [COLLIE:Snoopy] is not allowed because Snoopy is not a collie.

The join rule merges identical concepts. Two graphs may be joined by overlaying one graph on top of the other so that the two identical concepts merge into a single concept. As a result, all the conceptual relations that had been linked to either concept are linked to the single merged concept. Both the type label and the referent must be the same: [PERSON:Sam] may be joined to [PERSON:Sam] or [CAT] to [CAT]. If one concept is generic and the other is individual, they could be joined after restricting [PERSON] to [PERSON:Sam]. But no restriction could allow [PERSON:Sam] to be joined to [CAT] or [PERSON:Rock].

When two concepts are joined, some relations in the resulting graph may become redundant. One of each pair of duplicates can then be deleted by the rule of *simplification*: when two relations of the same type are linked to the same concepts in the same order, they assert the same information; one of them may therefore be erased. This rule corresponds to the rule of logic that $R(x,y) \wedge R(x,y)$ is equivalent to just $R(x,y)$. But the order of the arcs is significant: the formula $R(x,y) \wedge R(y,x)$ is not equivalent to $R(x,y)$.

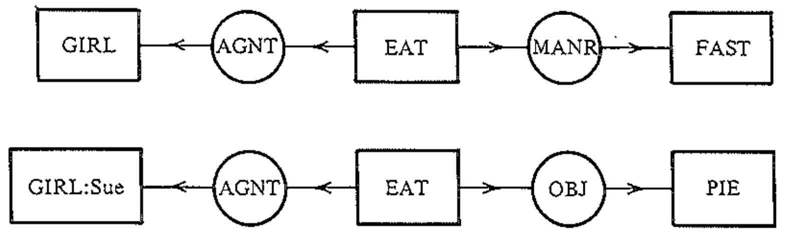

Fig. 3.8 Restriction of the second graph in Fig. 3.7

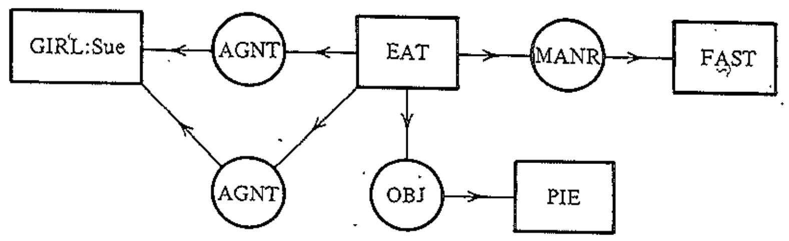

Fig. 3.9 Join of the two graphs in Fig. 3.8

3.4.2 Definition. Two conceptual relations of the same type are *duplicates* if for each *i*, the *i*th arc of one is linked to the same concept as the *i*th arc of the other.

To illustrate the formation rules, Fig. 3.7 shows two canonical graphs. The first one may be read *A girl is eating fast*; and the second, *A person, Sue, is eating pie.* If the concept of type PERSON in the second graph were restricted to type GIRL (by the rule of restriction), then the graphs of Fig. 3.7 would be changed to those in Fig. 3.8. But before doing the restriction, the conformity relation must be checked to ensure that GIRL::Sue is true.

Now the concept [GIRL] in the first graph can also be restricted to [GIRL:Sue]. The two identical pairs of concepts, [GIRL:Sue] and [EAT], can then be joined to each other. The result is Fig. 3.9. In that graph, the two copies of (AGNT) are duplicates. When one of them is deleted by simplification, the graph becomes Fig. 3.10, which may be read *A girl, Sue, is eating pie fast.*

The formation rules are a kind of *graph grammar* for canonical graphs. Besides defining syntax, they also enforce certain semantic constraints. Appendix B.3 shows the number and types of concepts expected for each conceptual relation. The rules can never relax or erase those constraints. If all the graphs in the starting set obey the constraints, then all the derived graphs will also obey them.

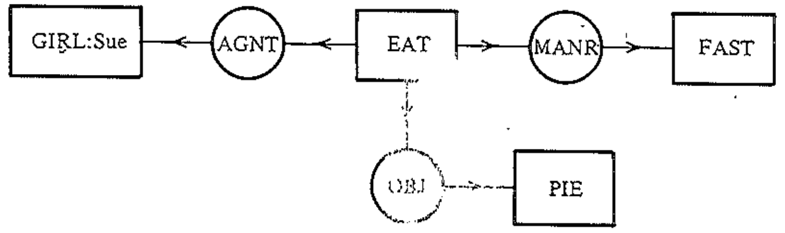

Fig. 3.10 Simplification of Fig. 3.9

3.4.3 Assumption. There are four *canonical formation rules* for deriving a conceptual graph w from conceptual graphs u and v (where u and v may be the same graph):

- *Copy.* w is an exact copy of u.
- *Restrict.* For any concept c in u, $type(c)$ may be replaced by a subtype; if c is generic, its referent may be changed to an individual marker. These changes are permitted only if $referent(c)$ conforms to $type(c)$ before and after the change.
- *Join.* If a concept c in u is identical to a concept d in v, then let w be the graph obtained by deleting d and linking to c all arcs of conceptual relations that had been linked to d.
- *Simplify.* If conceptual relations r and s in the graph u are duplicates, then one of them may be deleted from u together with all its arcs.

Although the number of canonical graphs may be infinite, the formation rules can derive all of them from a finite set. Suppose that the following two graphs are in the starting set:

```
[PHYSOBJ]→(ATTR)→[COLOR].
[SLEEP]→(AGNT)→[ANIMAL].
```

By restrict, the type label PHYSOBJ may be restricted to the subtype ANIMAL. Then join can merge the two concepts of type ANIMAL to form the graph,

```
[SLEEP]→(AGNT)→[ANIMAL]→(ATTR)→[COLOR].
```

This graph could be further restricted to form graphs for the sentences *A brown beaver sleeps* or *A purple cow sleeps*. But since IDEA is not a subtype of ANIMAL, there is no way of deriving *A green idea sleeps*. The formation rules enforce selectional constraints by preventing certain combinations from being derived.

Canonical formation rules are not *rules of inference*. If some girl is eating fast and Sue is eating pie, it does not follow that Sue is the one who is eating pie fast. The formation rules enforce selectional constraints, but they make no guarantee of truth or falsity. To see exactly what they do, consider the following levels of meaningfulness, adapted from Sommers (1959) and Odell (1971):

- Gibberish:
  Ozderst vwxo ahlazza.
- English words, but in an ungrammatical sequence:
  A am I number prime.
- Grammatical sequence, but violating selectional constraints:
  I am a prime number.

- Obeying selectional constraints, but violating rules of logic, meaning postulates, or word intensions:
  I am the prime minister of the U.K., and so is Margaret.
- Logically consistent, but possibly false:
  I am the prime minister of the U.K.
- Empirically true:
  I am writing about canonical formation rules in this section.

Syntactic rules can map noncanonical graphs into grammatical, but nonsensical sentences, such as *I am a prime number* or *Colorless green ideas sleep furiously*. Canonical graphs prevent such nonsense from being generated, but they may violate meaning postulates. To say that two people are prime minister at the same time is inconsistent with the meaning postulates for *prime minister*. If a canonical graph is not blocked by inconsistencies, it may be mapped to a logical sentence. To say *I am the prime minister* is logical and meaningful, but false. And if a graph describes an actual situation in the real world, the resulting sentence is empirically true.

The sentence *Colorless green ideas sleep furiously* contains several kinds of anomalies. The combinations *green ideas* and *ideas sleep* violate selectional constraints, and the combination *colorless green* violates rules of logic. The phrase *sleep furiously* does not violate logic or selectional constraints, but it is unexpected, unlikely, and implausible. That combination cannot be ruled out completely since one can imagine a person going to bed a with a dogged determination to spend a night of violent, unconscious tossing and turning. Instead, the *schemata* of Section 4.1 incorporate background knowledge about the world and the usual combinations of entities, attributes, and events in it. Since no schema would contain the combination *sleep furiously*, it would be considered unlikely, but not impossible. Canonical formation rules are a *context-free graph grammar*; they rule out nonsensical sentences with local constraints. Logic, however, requires *context-sensitive rules* that enforce global constraints. Such restrictions are enforced by the rules of inference in Chapter 4.

By means of joins and restrictions, the formation rules let a subtype inherit the properties of its supertype. Schank (1975) presented a list of fifteen primitive graphs from which all his other graphs could be derived by joins and restrictions. Even systems with a linear notation, such as KRL (Bobrow & Winograd 1977), rely on combining operations that have the same effect as joins. For relational databases, a join of two concepts is an intensional operation that parallels Codd's extensional joins (1970). If two database relations are described by conceptual graphs, the join of the two relations on a common domain is described by the join of the two graphs on a common concept.

3.4.4 Definition. Let A be any set of conceptual graphs. A graph w is said to be *canonically derivable* from A if either of the following conditions is true:

- w is a member of A.

- w may be derived by applying a canonical formation rule to graphs u and v that are themselves canonically derivable from A.

Schank's fifteen primitive graphs, from which all his other graphs are derivable, constitute a *canonical basis.* A canonical basis can be as small and simple as Schank's, or it can be a larger, richer set of graphs with more constraints and background information. Various trade-offs are possible. A small, simple basis contains very little knowledge of the world, and more information has to be packed into other structures, such as the schemata defined in Chapter 4.

3.4.5 Assumption. The *canon* contains the information necessary for deriving a set of canonical graphs. It has four components:

- A type hierarchy T,
- A set of individual markers I,
- A conformity relation :: that relates labels in T to markers in I,
- A finite set of conceptual graphs B, called a *canonical basis*, with all type labels in T and all referents either * or markers in I.

The canonical graphs are the *closure* of B under the canonical formation rules. If a new graph is canonized that cannot be canonically derived from B, then it must be added to B.

3.5 GENERALIZATION AND SPECIALIZATION

The canonical formation rules are *specialization rules.* Restriction, for example, specializes the concept [ANIMAL] to [DOG] or [BEAGLE]. Join specializes a graph by adding conditions and attributes from another graph. In the derivation from Fig. 3.7 to Fig. 3.10, the resulting graph has more conditions than either of the graphs from which it was derived, and the generic concept [GIRL] was specialized to the individual [GIRL:Sue]. Copy and simplification do not specialize a graph further, but neither do they generalize it.

This section considers the operation of *generalization*, which proceeds in the reverse order of specialization. Whereas specialization does not preserve truth, generalization does. If the girl Sue is eating pie fast, then it must be true that some girl is eating fast and that the person Sue is eating pie. Unfortunately, generalization does not necessarily preserve selectional constraints. If the girl Sue is eating pie, it follows that some entity is eating some entity, but the graph

```
[ENTITY]←(AGNT)←[EAT]→(OBJ)→[ENTITY]
```

does not include the constraints expected for the concept [EAT]. Even though generalizations are not necessarily canonical, they are important because they form the basis for logic and model theory in Chapter 4.

3.5.1 Definition. If a conceptual graph u is canonically derivable from a conceptual graph v (possibly with the join of other conceptual graphs $w_1,\ldots,w_n$), then u is called a *specialization* of v, written $u \leq v$, and v is called a *generalization* of u.

This definition has important implications: any graph is a generalization of itself; any subgraph is a generalization of the original; replacing a type label with a supertype generalizes a graph; and erasing an individual marker generalizes a graph. In particular, the graph consisting of the single concept [⊤] is a generalization of every other conceptual graph. These properties are proved as the next theorem.

3.5.2 Theorem. Generalization defines a partial ordering of conceptual graphs called the *generalization hierarchy.* For any conceptual graphs u, v, and w, the following properties are true:

- *Reflexive.* $u \leq u$.
- *Transitive.* If $u \leq v$ and $v \leq w$, then $u \leq w$.
- *Antisymmetric.* If $u \leq v$ and $v \leq u$, then $u = v$.
- *Subgraph.* If v is a subgraph of u, then $u \leq v$.
- *Subtypes.* If u is identical to v except that one or more type labels of v are restricted to subtypes in u, then $u \leq v$.
- *Individuals.* If u is identical to v except that one or more generic concepts of v are restricted to individual concepts of the same type, then $u \leq v$.
- *Top.* The graph [⊤] is a generalization of all other conceptual graphs.

Proof. Since u is canonically derivable from itself by the copy rule, $u \leq u$. If u is canonically derivable from v and v is canonically derivable from w, then u must be canonically derivable from w; therefore, $u \leq w$. If u is canonically derivable from v and v is canonically derivable from u, the only way they could have been derived is by copy; therefore, they must be identical. If v is a subgraph of u, then u must be canonically derivable from v by joining the other parts of u that are not included in v; therefore, $u \leq v$. If u is derived from v by restricting type labels to subtypes or generic concepts to individual, then u is canonically derivable from v; therefore, $u \leq v$. Finally, any graph u can be canonically derived from [⊤] (plus some other graph) simply by letting the other graph be u itself; then restrict [⊤] to the type and referent of any concept c in u and join it to c.

If the graph u is a specialization of v, whenever u represents a true situation, v must also represent a true situation. In Section 4.3, that property is shown directly by inferences on conceptual graphs. Another proof would use the operator ϕ that translates conceptual graphs into logical formulas. The next theorem shows that if $u \le v$, a canonical derivation of u from v corresponds to the reverse of a proof of the formula ϕv from the formula ϕu. The proof depends on the fact that the formation rules add properties to a graph. But if A and B are any properties, then $(A \wedge B) \supset A$. Hence the graph u with more properties implies the simpler graph v.

3.5.3 Theorem. For any conceptual graphs u and v, if $u \le v$, then $\phi u \supset \phi v$.

Proof. Consider a canonical derivation of u from v with intermediate graphs $v_1, v_2, \ldots, v_n$ where $v = v_1$ and $u = v_n$. To prove that ϕu implies ϕv, show that at each step ϕv_{i+1} implies ϕv_i. Then the sequence of formulas $\phi v_n, \ldots, \phi v_1$ would constitute a proof of ϕv under the hypothesis of ϕu. The rule for deriving the graph v_{i+1} from v_i must be either copy, simplify, restrict, or join:

- If copy, v_{i+1} is identical to v_i. Therefore, ϕv_{i+1} implies ϕv_i.
- If simplify, ϕv_i contains a duplicate predicate that is omitted in ϕv_{i+1}. Since any formula A implies the conjunction $A \wedge A$, ϕv_{i+1} implies ϕv_i.
- If a type label T is restricted to a subtype S, ϕv_i has a predicate $T(x)$ that is replaced by a predicate $S(x)$ in ϕv_{i+1}. By Assumption 3.2.4, $\delta S \subseteq \delta T$. Hence for any x, $S(x)$ implies $T(x)$. If a generic marker is restricted to an individual i, then $S(i)$ implies the generic $\exists x S(x)$. In either case, ϕv_{i+1} implies ϕv_i.
- If join, ϕv_{i+1} is equivalent to a formula of the form $\exists x_1 \ldots \exists x_k (P \wedge Q \wedge x{=}y)$ where P is the body of ϕv_i, Q is a conjunction of predicates derived from some other graph w that was joined to v_i, and the equation $x{=}y$ equates the two identifiers of the concepts that were joined. But the conjunction $P \wedge Q \wedge x{=}y$ implies P. Therefore, ϕv_{i+1} implies ϕv_i.

The canonical formation rules are the opposite of rules of inference. Whereas a rule of inference derives true graphs from true graphs, the canonical formation rules derive false graphs from false graphs. For this reason, they may be called *refutation rules*: one way to refute a graph is to show that it is canonically derivable from a false graph. This property is used in the *open-world models* of Section 4.5: two special sets of graphs T and F are introduced, where all graphs in T are true and all graphs in F are false. If u is a generalization of some graph in T, it must also be true; but if u is a specialization of some graph in F, it must be false.

If u is a specialization of v, there must be a subgraph u' embedded in u that represents the original v to which additional graphs were joined during the canonical derivation. That subgraph u' is called a *projection* of v in u. The Greek letter π is used for a *projection operator*: $u' = \pi v$. Every conceptual relation in πv must be identical to the corresponding relation in v, but some of the concepts in v may have been restricted to subtypes or may have been converted from generic to individual. In the derivation of u from v, some concepts of v may have been joined to each other, and some conceptual relations may have been eliminated as duplicates. Therefore, the projection πv must contain a basic core of v, but its shape and concept types may be different. In fact, v may be a planar, acyclic graph, but πv might be folded back upon itself to form complex cycles.

3.5.4 Theorem. For any conceptual graphs u and v where $u \le v$, there must exist a mapping $\pi: v \to u$, where πv is a subgraph of u called a *projection* of v in u. The *projection operator* π has the following properties:

- For each concept c in v, πc is a concept in πv where $type(\pi c) \le type(c)$. If c is individual, then $referent(c) = referent(\pi c)$.
- For each conceptual relation r in v, πr is a conceptual relation in πv where $type(\pi r) = type(r)$. If the ith arc of r is linked to a concept c in v, the ith arc of πr must be linked to πc in πv.

The mapping π is not necessarily one-to-one: if x_1 and x_2 are two concepts or conceptual relations where $x_1 \ne x_2$, it may happen that $\pi x_1 = \pi x_2$. The mapping π is not necessarily unique: the graph v may also have another projection $\pi' v$ in u where $\pi' v \ne \pi v$.

Proof. Construct the mapping π as the composition of separate mappings $\pi_1, \ldots, \pi_n$ that correspond to the steps of some canonical derivation of u from v. Consider step i of the derivation:

- If copy, let π_i be the identity mapping.
- If restriction of a concept c, let π_i map c to the newly restricted form, and let it be the identity mapping on the rest of the graph. For π_i, the conclusions of the theorem hold.
- If join, let π_i map both concepts that were joined to the single joined concept, and let it be the identity mapping on the rest of the graph. Since joins do not change referents and they keep relations linked to corresponding concepts, the conclusions must hold.
- If simplify, let π_i map both duplicate relations into the single conceptual relation that remains, and let it be the identity mapping on the rest of the graph. Since duplicate relations, by definition, must be of the same type and be linked to exactly the same concepts in exactly the same order, the conclusions of the theorem must hold.

Let πv be the composition $\pi_n \ldots \pi_1 v$. Since each step π_i preserves the conditions and conclusions of the theorem, the composition π must also preserve them.

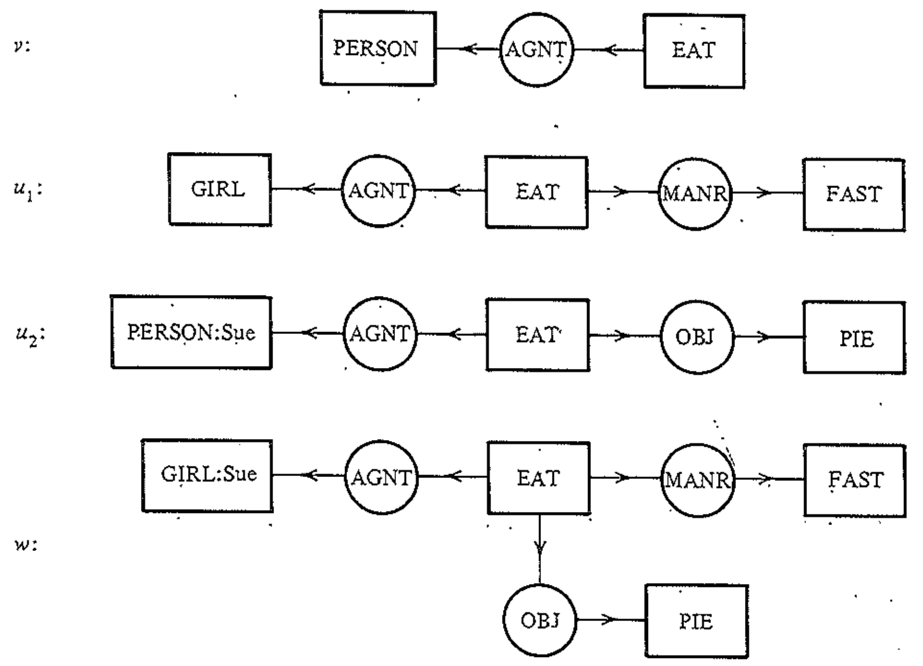

Fig. 3.11 A common generalization and a common specialization

Projections map graphs at higher levels of the generalization hierarchy into ones at lower levels. The hierarchy is not a lattice because two graphs may not have a unique minimal common generalization and a unique maximal common specialization. But any two graphs have at least one *common generalization*, since the graph [⊤] is a common generalization of all. There is no guarantee that they must have a *common specialization*, but many of them do.

3.5.5 Definition. Let u_1, u_2, v, and w be conceptual graphs. If $u_1 \leq v$ and $u_2 \leq v$, then v is called a *common generalization* of u_1 and u_2. If $w \leq u_1$ and $w \leq u_2$, then w is called a *common specialization* of u_1 and u_2.

Whenever two graphs u_1 and u_2 are joined on one or more concepts, the resulting graph w is a common specialization of both. In Fig. 3.11, the two middle graphs were joined to form the lower graph, which is a common specialization. Whenever two graphs are joined in such a way, the parts that were merged must be projections of some common generalization. In this example, they are both projections of the top graph v. Conversely, if two graphs u_1 and u_2 have a common generalization v, then the corresponding projections $\pi_1 v$ in u_1 and $\pi_2 v$ in u_2 are candidates for being merged

by a series of joins (possibly with additional restrictions and simplifications). Such a merger might be blocked, however, by incompatible type labels or referents. If there are no incompatibilities, then the two projections are said to be *compatible*.

3.5.6 Definition. Let conceptual graphs u_1 and u_2 have a common generalization v with projections π_1: $v \rightarrow u_1$ and π_2: $v \rightarrow u_2$. The two projections are said to be *compatible* if for each concept c in v, the following conditions are true:

- $type(\pi_1 c) \cap type(\pi_2 c) > \bot$.
- The referents of $\pi_1 c$ and $\pi_2 c$ conform to $type(\pi_1 c) \cap type(\pi_2 c)$.
- If $referent(\pi_1 c)$ is the individual marker i, then $referent(\pi_2 c)$ is either i or $*$.

The common specialization w may be derived by joining the graphs u_1 and u_2 on compatible projections of the more general graph v. A good way to visualize the join is to imagine the graphs u_1 and u_2 drawn on plastic transparencies and then overlaid on each other so that the compatible projections merge (some stretching and twisting may be necessary). Since any graph can be projected into one of its specializations, both u_1 and u_2 can be projected into w. For Fig. 3.11, the subgraph of w that falls in the overlap of u_1 and u_2 is the following projection of v into w:

```
[GIRL:Sue]←(AGNT)←[EAT].
```

The conditions for compatible projections ensure that corresponding concepts can be restricted to identical forms and then be joined. Definition 3.5.6 rules out a restriction to the absurd type ⊥ since no possible entity could ever conform to type ⊥. If all the conditions of 3.5.6 are satisfied, the two graphs can be merged by a *join on compatible projections*.

3.5.7 Theorem. If conceptual graphs u_1 and u_2 have a common generalization v with compatible projections π_1: $v \rightarrow u_1$ and π_2: $v \rightarrow u_2$, then there exists a unique conceptual graph w with the following properties:

- w is a common specialization of u_1 and u_2.
- There exist projections π_1': $u_1 \rightarrow w$ and π_2': $u_2 \rightarrow w$ where $\pi_1' \pi_1 v = \pi_2' \pi_2 v$.
- If w' is any other conceptual graph with the above two properties, then $w' < w$.

The graph w is called a *join on compatible projections* of u_1 and u_2. If both u_1 and u_2 are canonical graphs, then so is w.

Proof. The graph w can be constructed from u_1 and u_2 by the following canonical derivation:

- For each concept c_i in v, restrict $\pi_1 c_i$ and $\pi_2 c_i$ to $type(\pi_1 c_i) \cap type(\pi_2 c_i)$.

- For each concept c_i in v, join the newly restricted concepts $\pi_1 c_i$ and $\pi_2 c_i$ to each other.
- Simplify the resulting graph by eliminating duplicate relations.

The definition of compatible projections ensures that these restrictions and joins are permissible. Therefore, w is canonical if u_1 and u_2 are canonical. To show that w is a generalization of any other common specialization of u_1 and u_2 that meets the condition $\pi_1{}'\pi_1 v = \pi_2{}'\pi_2 v$, observe that the construction of w does the minimum number of restrictions, joins, and simplifications necessary to meet that condition. Therefore, any other common specialization meeting that condition must be derivable from w by additional applications of the canonical formation rules.

Since two conceptual graphs may have many different common generalizations, they may also have many different pairs of compatible projections. For most computations, large compatible projections are preferred to smaller ones. The next theorem gives the conditions for one pair of compatible projections to be an *extension* of some other pair.

3.5.8 Theorem. Let conceptual graphs u_1 and u_2 have a common generalization v with compatible projections π_1: $v \rightarrow u_1$ and π_2: $v \rightarrow u_2$, and let v' be a proper subgraph of v. Then v' is also a common generalization of u_1 and u_2 with compatible projections π_1: $v' \rightarrow u_1$ and π_2: $v' \rightarrow u_2$. The compatible projections $\pi_1 v$ and $\pi_2 v$ are said to be *extensions* of $\pi_1 v'$ and $\pi_2 v'$.

Proof. Since v' is a subgraph of v, $v \leq v'$. Since $u_1 \leq v$ and $u_2 \leq v$, it follows that $u_1 \leq v'$ and $u_2 \leq v'$. Since $\pi_1 v$ and $\pi_2 v$ are compatible, their subgraphs $\pi_1 v'$ and $\pi_2 v'$ must also be compatible.

If two graphs contain compatible projections of a common generalization v, those projections might be extended by finding a larger common generalization that includes v as a subgraph. Since all conceptual graphs are finite, the process of extension must eventually stop. When it stops, the resulting compatible projections are called *maximally extended*. A join on those projections is then called a *maximal join*. The join is locally maximal, because there may be other compatible projections of two graphs that are also maximally extended.

One algorithm for computing a maximal join is to start by joining two graphs on a single concept. Then extend the join by one conceptual relation at a time by looking for potential candidates along the boundaries of the part that has already been joined. When no more candidates can be found, the join is locally maximal. For unlabeled graphs, finding a globally maximal join is a difficult combinatorial problem. For practical applications, such as comparing organic molecules, analyzing

scene descriptions, or joining conceptual graphs, the labels on the nodes reduce the combinations to a manageable number. McGregor (1982) presents efficient algorithms for computing maximal joins.

3.6.9 Definition. Two compatible projections are said to be *maximally extended* if they have no extensions. A join on maximally extended compatible projections is called a *maximal join.*

Maximal joins are important because they join two graphs on maximally connected subparts. They form the basis for *preference semantics* (Wilks 1975), which encourages maximum connectivity in the generated graphs. The mechanisms of type expansion in Section 3.6, schematic join in Section 4.1, heuristic preference rules in Section 4.7, semantic interpretation in Section 5.6, and database inference in Section 6.5 are all based on maximal joins.

3.6 ABSTRACTION AND DEFINITION

Verbs like *think, know,* and *believe* take complete sentences as their objects. They lead to structures of graphs embedded inside the nodes of other graphs. The statement *Norma loves Joe* can be represented in a simple graph, but the statement *I think that Norma loves Joe* leads to the graph,

```
[PROPOSITION: [PERSON:Norma]←(EXPR)←[LOVE]→(OBJ)→[PERSON:Joe]]
```

where the object of [THINK] is a concept with type label PROPOSITION and the graph of *Norma loves Joe* as its referent. One of the most important verbs that takes a sentence as its object is *define.* A definition can equate an entire graph with a name or type label.

Some systems do not support definitions. For MARGIE, Schank (1975) claimed that only eleven primitive types of acts were sufficient to represent a wide range of English. For acts such as BUY, SELL, BEAT, or RACE, the parser for MARGIE translated the complex act into a combination of primitives (Riesbeck 1975). Some systems are not dogmatic about which concepts are primitive, but they have no mechanisms for dynamically defining new types in terms of more primitive ones.

Type definitions provide a way of expanding a concept in primitives or contracting a concept from a graph of primitives. Even though Schank did not have a formal mechanism for definitions, Riesbeck's parser did expansions upon input. For output, Goldman's language generator (1975) contracted large graphs into single verbs like *admit* or *threaten.* Without a formal theory of definitions, the input parser and the output generator used different mechanisms, and Rieger's inference engine (1975) had to do its reasoning upon large graphs expanded in low-level primitives. More recently, Schank and his colleagues have begun to introduce high-level types such as

AUTHORIZE, ORDER, and PETITION (Schank & Carbonell 1979) or KISS, MOAN, and SHOUT (Rieger 1979). Then a single concept like [KISS] can trigger inferences without a search for patterns of passionate lip contact.

Definitions can specify a type in two different ways: by stating necessary and sufficient conditions for the type, or by giving a few examples and saying that everything similar to these belongs to the type. The first method derives from Aristotle's method of definition by *genus* and *differentiae*, and the second is closer to Wittgenstein (1953). AI systems have supported both methods:

- Definitions by genus and differentiae are logically easiest to handle. They have been incorporated in REL (Thompson & Thompson 1975), OWL (Martin 1979), and MCHINE (Ritchie 1980).
- Definitions by examples or prototypes are essential for dealing with natural language and its applications to the real world, but their logical status is unclear. Some systems that support prototypes are KRL (Bobrow & Winograd 1977), KL-ONE (Brachman 1979), and TAXMAN (McCarty & Sridharan 1981).

TAXMAN is designed for legal reasoning in corporate tax law. Since most legal disputes arise when the issues are vague, ill-defined, or otherwise difficult to classify, definitions by necessary and sufficient conditions are not possible. For greater flexibility, TAXMAN uses prototypes that represent the standard cases and *deformations* that adapt the prototypes to changing conditions.

Conceptual graphs support type definitions by genus and differentiae as well as *schemata* and *prototypes*, which specify sets of family resemblances. Both methods are based on *abstractions*, which are canonical graphs with one or more concepts designated as *formal parameters*.

3.6.1 Definition. An *n-adic abstraction*, $\lambda a_1,...,a_n u$, consists of a canonical graph u, called the *body*, together with a list of generic concepts $a_1,...,a_n$ in u, called the *formal parameters*. The *parameter list* following λ distinguishes the formal parameters from the other concepts in u.

An abstraction is like a procedure in a programming language. The Greek letter λ introduces the formal parameters. In the body, each concept used as a parameter contains one of the variable symbols in its referent field:

```
λx,y [SUPPLY]-
        (AGNT)→[SUPPLIER:*x]
        (OBJ)→[PART:*y]→(COLR)→[RED].
```

In this example, λx,y identifies [SUPPLIER:*x] and [PART:*y] as formal parameters. The concepts [SUPPLY] and [RED], which are not parameters, are like local variables in a procedure or function. The body of an abstraction is a conceptual graph that asserts some proposition. When n formal parameters are identified, the abstraction becomes an n-adic predicate, which is true or false only when specific referents are assigned to its parameters. The formula operator ϕ maps abstractions into λ-expressions in standard logic; each such expression defines a new predicate. See Appendix A.2 for a discussion of the lambda calculus.

3.6.2 Assumption. The formula operator ϕ maps n-adic abstractions into n-adic lambda expressions:

- Let $\lambda a_1,...,a_n u$ be an n-adic abstraction.
- Let $x_1,...,x_m$ be variables assigned to the generic concepts of u other than the formal parameters.
- Remove the quantifiers from the formula ϕu to leave the predicates Φ.

Then $\phi\lambda a_1,...,a_n u$ is the lambda expression, $\lambda a_1,...,a_n \exists x_1...\exists x_m \Phi$.

For the abstraction relating suppliers and parts in the previous example, the formula operator ϕ would generate the following λ-expression in standard logic:

$$\lambda x,y \exists z \exists w(\mathrm{SUPPLY}(z) \wedge \mathrm{AGNT}(z,x) \wedge \mathrm{SUPPLIER}(x) \wedge \mathrm{OBJ}(z,y) \wedge \mathrm{PART}(y) \wedge \mathrm{COLR}(y,w) \wedge \mathrm{RED}(w)).$$

In some database systems, a lambda expression is used as a query specification. This expression corresponds to the query *List suppliers and parts where the suppliers supply the parts and the parts are colored red.* For that query, the clause that occurs after *where* is mapped to the body of the expression, and the words between *list* and *where* are mapped to the formal parameters. For many queries, the terms marked by question words like *who, which*, or *what* become the formal parameters. To answer the query, the denotation operator δ searches the database for all values of suppliers and parts that make the relationships in the graph true. The denotation of an abstraction is a set of n-tuples (in this case, pairs of SUPPLIER and PART) that may be assigned as the referents of the formal parameters to make the graph true.

The generalization hierarchy of conceptual graphs extends to abstractions. One abstraction is a specialization of another if its body is a specialization of the other's body. There is an additional constraint, however: the projection π from the more general graph to the more specialized one must map corresponding parameters to each other. The hierarchy of abstractions is similar to the hierarchy of procedures in the TAXIS system for database semantics (Mylopoulos et al. 1980).

3.6.3 Assumption. A generalization hierarchy is defined over abstractions. For a pair of n-adic abstractions, $\lambda a_1,...,a_n u \leq \lambda b_1,...,b_n v$ if the following conditions hold:

- For the two bodies, $u \leq v$.
- There exists a projection π of v into u, which for all i maps the parameter b_i of v into the parameter a_i of u.

type KISS(x) is

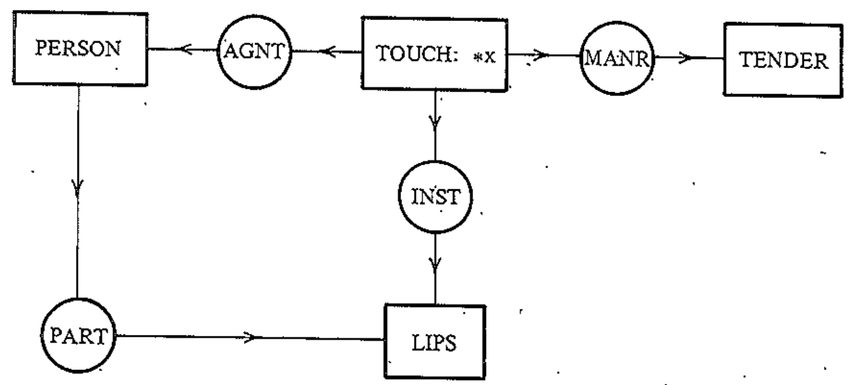

Fig. 3.12 Type definition for KISS

New type labels are defined by an Aristotelian approach. Some type of concept is named as the genus, and a canonical graph, called the differentia, distinguishes the new type from the genus. The differentia is the body of a monadic abstraction, and the genus is the type label of the formal parameter. As an example of type definition, Fig. 3.12 defines KISS with genus TOUCH and with a differentia graph that says that the touching is done by a person's lips in a tender manner. Since the keyword **type** introduces the type label and the parameter list, the symbol λ may be omitted. Type definition is like function definition in LISP: a method for assigning a label to an abstraction. As in LISP, unlabeled abstractions can appear anywhere that a type label might appear, as in the type field of a concept box or relation circle.

3.6.4 Assumption. A *type definition* declares that a type label t is defined by a monadic abstraction $\lambda a\ u$. It is written, **type** $t(a)$ **is** u. The body u is called the *differentia* of t, and *type*(a) is called the *genus* of t. The abstraction $\lambda a\ u$ may be written in the type field of any concept where the type label t may be written.

Any generic concept in a canonical graph may be chosen as the genus of a type definition. Following are three type definitions based on the same differentia, but with different concepts marked as the genus. The first example defines the label CIRCUS-ELEPHANT as a subtype of ELEPHANT that performs in a circus:

type CIRCUS-ELEPHANT(x) **is**
    [ELEPHANT:*x]←(AGNT)←[PERFORM]→(LOC)→[CIRCUS].

Either of the other two concept types in this graph could have been chosen as the genus instead of [ELEPHANT]. If [CIRCUS] had been marked as the formal parameter, it would define a type of CIRCUS that had a performing elephant:

type ELEPHANT-CIRCUS(y) **is**
    [ELEPHANT]←(AGNT)←[PERFORM]→(LOC)→[CIRCUS:*y].

If [PERFORM] had been marked as the parameter, it would define a type of performance that an elephant does in a circus:

type ELEPHANT-PERFORMANCE(z) **is**
    [ELEPHANT]←(AGNT)←[PERFORM:*z]→(LOC)→[CIRCUS].

This definition implies that an elephant performance must have at least one elephant, but it does not rule out the possibility of multiple elephants. To be more explicit, the notation [ELEPHANT:{*}], introduced in Section 3.7, could be used to represent an arbitrary set of elephants.

An important use for type definition is to describe a *subrange* that limits the possible referents for a concept. The type POSITIVE, for example, could be defined as a number that is greater than zero:

type POSITIVE(x) **is** [NUMBER:*x]→(>)→[NUMBER:0].

To avoid the need for defining a special type label for every subrange, the abstraction that defines a type may be used in the type field of a concept without associating it with a particular label. In the following concept, the λ-expression in the type field constrains the referent to be a positive number, in this case, 15:

[λx [NUMBER:*x]→(>)→[NUMBER:0]: 15].

An even shorter notation is "NUMBER>0" as an abbreviation for the λ-expression. The string ">0" is treated as part of the type field. Since 15 is a positive number, the concept [NUMBER>0:15] is well formed. Similar subranges are 0<NUMBER<25 or SPEED≤55mph. The latter is an abbreviation for the following abstraction:

λx [SPEED:*x]→(MEAS)→[MEASURE]→(≤)→[MEASURE:55mph].

Once a mechanism is available for defining new types, the definitions can be used to simplify the graphs. Type contraction deletes a complete subgraph and incorporates the equivalent information in the type label of a single concept. If some graph u happens to contain a subgraph u' that corresponds to the body of some type definition $t=\lambda av$, then redundant parts of u' may be deleted. In its place, the concept of u' that corresponds to the parameter a of v has its type label replaced with t. The next definition specifies an algorithm for carrying out the contraction.

3.6.5 Assumption. Let u be a canonical graph, and let type t be defined as $\lambda a v$. If u is a specialization of v, π is a projection of v into u, and $type(\pi a)=type(a)$, then the operation of *type contraction* may be performed on u by the following algorithm:

```
replace the type label of $\pi a$ with $t$;
leave $referent(\pi a)$ unchanged;
for $b$ in the concepts and conceptual relations of $v$ where
      $b \neq a$, $\pi b$ identical to $b$, and $\pi b$ not a cutpoint of $u$ loop
   if $b$ is a concept then
      detach $\pi b$ from $u$;
   else
      detach $\pi b$ and all its arcs from $u$;
   end if;
end loop;
for $e$ in the arcs left in $u$ not linked to a concept loop
   reattach the concept that had been linked to arc $e$ in $u$;
end loop;
```

If type contraction is performed on a canonical graph, the resulting graph is also canonical. The notation $[\lambda a\ v{:}\ i]$ represents the contracted form of the concept πa, where the type is $\lambda a\ v$, and the referent i is the original $referent(\pi a)$.

As an example of type contraction, let u be the following graph, which may be read *The elephant Clyde, who performs in a circus, is gray,*

```
[GRAY]←(COLR)←[ELEPHANT:Clyde]←(AGNT)←[PERFORM]→(LOC)→[CIRCUS].
```

This graph corresponds to graph u in Assumption 3.6.5, and v is the differentia for defining CIRCUS-ELEPHANT:

```
[ELEPHANT:*x]←(AGNT)←[PERFORM]→(LOC)→[CIRCUS].
```

The following graph is the projection πv of v into u:

```
[ELEPHANT:Clyde]←(AGNT)←[PERFORM]→(LOC)→[CIRCUS].
```

The concept [ELEPHANT:Clyde] is πa, which is a projection of the genus concept [ELEPHANT:*x] of v. The next stage of type contraction replaces the type label of πa to form the graph,

```
[GRAY]←(COLR)←[CIRCUS-ELEPHANT:Clyde]-
     (AGNT)←[PERFORM]→(LOC)→[CIRCUS].
```

The **for**-loop in Assumption 3.6.5 now begins to detach concepts and conceptual relations from this graph. In the first iteration of the loop, the only candidate for πb is the concept [CIRCUS]. After πb is detached, the relation (LOC) is left with a dangling arc:

```
[GRAY]←(COLR)←[CIRCUS-ELEPHANT:Clyde]-
     (AGNT)←[PERFORM]→(LOC)→
```

When the **for**-loop is repeated, the next candidate for πb is (LOC), which is then detached together with its arcs. Then [PERFORM] is detached, and finally (AGNT) is detached to generate the following graph:

```
[GRAY]←(COLR)←[CIRCUS-ELEPHANT:Clyde].
```

This graph, which may be read *The circus elephant Clyde is gray*, is the final result of type contraction, since no conceptual relations with dangling arcs are left.

Type contraction deletes subgraphs that can be recovered from information in the differentia. If some nodes covered by the differentia are linked to other nodes that are not covered, then they cannot be deleted. Otherwise, the graph would become disconnected. The inverse of type contraction is *type expansion*, which replaces a concept type with its definition. The type label of the genus replaces the defined type label, and the graph for the differentia is joined to the concept.

3.6.6 Definition. Let u be a canonical graph containing a concept a where $type(a)=\lambda b\ v$. Then *minimal type expansion* consists of joining the graphs u and v on the concepts a and b.

A minimal type expansion joins the differentia v to the graph u, but it does not replace the type label of the concept a. The result is canonical because it could be derived simply by restricting b to $type(a)$ and then doing a join. Yet the result of type contraction followed by a minimal type expansion is not identical to the original. It may contain a redundant subgraph, and the type label of the concept a is not restored. A *maximal type expansion*, however, makes more extensive changes to restore the graph to the original as far as possible.

3.6.7 Definition. A *maximal type expansion* starts with a minimal type expansion and takes the following additional steps. Let a, b, u, and v satisfy the same hypotheses as in Definition 3.6.6:

- Extend the join of a and b to a maximal join.
- Replace the type label of concept a with a type label t, where $type(a) \le t \le type(b)$, the result of replacing $type(a)$ with t is canonical, and there is no type s where $t < s \le type(b)$ and the result of replacing $type(a)$ with s would be canonical.

The more conservative, minimal expansion preserves truth: if the original graph is true, the expanded graph must also be true. A maximal expansion merges the new parts with the old parts as far as possible; yet the merger is only plausible, not logically certain. To illustrate type expansion, Fig. 3.13 defines BUY as a type of TRANSACTION. The transaction has two components, each an instance of GIVE.

type BUY(x) is

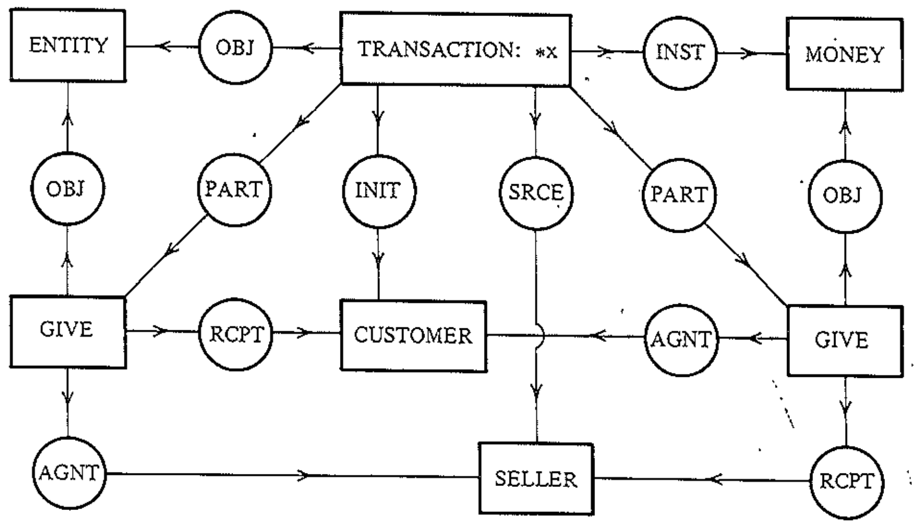

Fig. 3.13 Type definition for BUY

The initiator of the transaction is a CUSTOMER who is the recipient of one act of GIVE and the agent of the other GIVE. The object of the transaction is an ENTITY that is also the object of the first GIVE. The instrument of the transaction is MONEY, which is also the object of the second GIVE. And the source of the transaction is a SELLER who is the agent of one GIVE and the recipient of the other.

Now suppose that the phrase *Joe buying a necktie from Hal for $10* were translated to Fig. 3.14. The concepts [BUY] and [NECKTIE] are generic concepts, [PERSON:Joe] and [PERSON:Hal] are individual concepts, and [MONEY:@$10] is a measure contraction for,

```
[MONEY]→(MEAS)→[MEASURE]→(NAME)→["$10"].
```

The contracted form follows the conventions of Section 3.4 for units of measure. The @ symbol shows that $10 is a measure of money, not the name of some money.

By type expansion, [BUY] in 3.14 would be joined to [TRANSACTION] in 3.13. When the join is extended to a maximal join, the result is Fig. 3.15, in which the generic concepts [CUSTOMER], [SELLER], and [MONEY] have new referents assigned. The concept [NECKTIE] is still generic because no specific necktie was mentioned, but it is more restricted than [ENTITY].

Type expansions should be distinguished from the expansions done by Riesbeck's parser (1975). Instead of generating an intermediate graph like 3.14, Riesbeck translated directly to the primitives. Furthermore, he would generate the same graph

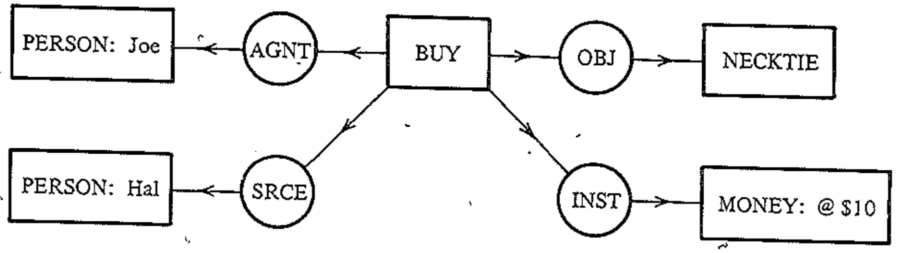

Fig. 3.14 Graph for "Joe buying a necktie from Hal for $10"

for the sentence *Joe bought a necktie from Hal for $10*, its converse *Hal sold Joe a necktie for $10*, and the pair of sentences *Joe gave Hal $10* and *Hal gave Joe a necktie.* Yet generating a single graph for all these sentences loses information about the initiator of the transaction and about the causal relationships. The two acts of giving, for example, might be an exchange of Christmas presents, where notions of buying or selling do not apply. The type definition in Fig. 3.13 explicitly shows that buying is a transaction with two separate acts of giving. Selling would have the same two acts of giving, but with a different initiator. Exchanging Christmas presents would be a different kind of transaction.

In Section 3.2, the type hierarchy was introduced as an *a priori* assumption. If the type labels are specified by type definitions, Assumption 3.6.8 relates the partial

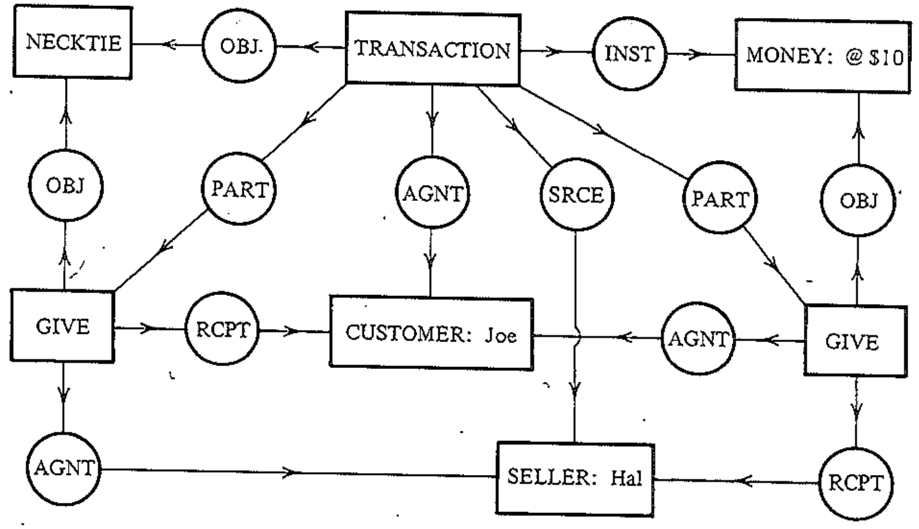

Fig. 3.15 Result of type expansion of Fig. 3.14

ordering to the structure of the definitions. This assumption states the Aristotelian principle that adding more differentiae to a concept restricts it to a lower subtype. Commutativity of type contractions is another common principle: if the type label CIRCUS-ELEPHANT is defined by ELEPHANT which performs in a circus and GRAY-ELEPHANT by ELEPHANT which is gray, then the type label GRAY-CIRCUS-ELEPHANT defined by ELEPHANT which is gray and performs in a circus should be equivalent to CIRCUS-ELEPHANT which is gray and to GRAY-ELEPHANT which performs in a circus.

3.6.8 Assumption. If type t is defined as $\lambda a\ u$, the position of t in the type hierarchy is determined by the following conditions:

- If the graph u consists of the single concept a, then $t{=}type(a)$.
- If u is larger than than the single concept a, then $t{<}type(a)$.
- Type contraction is commutative: if the graph u is derivable by joining canonical graphs v and w on the concept a, then $\lambda a\ u = \lambda[\lambda a\ v]w = \lambda[\lambda a\ w]v$.

An Aristotelian type hierarchy is one where the subtypes are determined by explicit type definitions. For Aristotle himself, there were nine or ten primitive types that had no definitions: SUBSTANCE, QUANTITY, QUALITY, RELATION, TIME, POSITION, STATE, ACTIVITY, and PASSIVITY. For the OWL system, the only primitive type is SUMMUM-GENUS, which corresponds to $\top$. All other types are introduced by definitions.

3.6.9 Definition. A type hierarchy T is said to be *Aristotelian* if every type label t that is a proper subtype of another type label is defined by an abstraction $t = \lambda a\ u$.

When all types other than the primitives are introduced by definitions, the partial ordering is completely determined. Furthermore, if one type is a subtype of another, then there must be some graph that states the differentia between the subtype and the supertype. That graph may, in fact, be the join of a large number of graphs if there are many intervening levels in the hierarchy.

3.6.10 Theorem. In an Aristotelian type hierarchy, if a type label s is a proper subtype of t ($s{<}t$), then there exists a type definition $s{=}\lambda a\ u$, where $type(a){=}t$. The graph u is called the *differentia* between s and t.

Proof. Since $s{<}t$, there must exist some type definition $s{=}\lambda a_1 u_1$. Then either $type(a_1){=}t$ or else by Assumption 3.6.8 there exists a sequence of type definitions $\lambda a_2 u_2,\ldots,\lambda a_n u_n$ where $type(a_i){=}\lambda a_{i+1}u_{i+1}$, for each i from 1 to n-1, and $type(a_n){=}t$. Then $s{=}\lambda[\lambda[\ldots[\lambda a_n u_n]\ldots]u_2]u_1$ and the differentia u is the join of the graphs u_1 through u_n on a_n.

In an Aristotelian type hierarchy, the rule of restricting a type label is redundant. If all subtypes are introduced by type definition, then restriction is equivalent to a join followed by a type contraction. The next theorem proves this fact.

3.6.11 Theorem. If the type hierarchy is Aristotelian, then any graph that is canonically derivable by restricting a type label to a subtype could also be derived by a join followed by a type contraction.

Proof. Let u be a graph derived from a canonical graph v by restricting a concept a in v to a subtype t. By Theorem 3.6.10, there exists a type definition $t{=}\lambda b.w$ where $type(b){=}type(a)$. Then join concept b of w to a of v. Since a is a cutpoint of the resulting graph, it is possible to do a type contraction by removing the graph w that had just been joined to v and replacing the type label of a with t. The result is u.

An Aristotelian type hierarchy is possible only for the artificially constructed types of a programming language. In the fields of science, accounting, or law, the practitioners strive to develop complete definitions for all concepts. But as long as those fields are growing, that goal can never be achieved. In ordinary language, very few concepts have complete definitions. Some types may be specified by definitions, but most are simply used without definitions. One reason for not defining everything is that present knowledge may be incomplete. A person may know that WOMBAT <MARSUPIAL, but may not have any other information for distinguishing a wombat from a kangaroo or a koala bear. A systems analyst may want to leave the type hierarchy partially undefined as part of a top-down design method. Undefined types are place holders that are filled in as the design progresses. The most compelling reason for not defining all type labels is that such definitions are impossible. Entities in the real world are not arranged in a neat hierarchy with precisely defined boundaries. GAME, for example, is a subtype of ACTIVITY, yet as Wittgenstein (1953) pointed out, different kinds of games bear a family resemblance to one another, but no complete definition by genus and differentiae is possible.

New conceptual relations may also be defined by abstractions. The next definition introduces the monadic relation (PAST) for past tenses:

```
relation PAST(x) is
     [SITUATION: *x]→(PTIM)→[TIME]→(SUCC)→[TIME: Now].
```

According to this graph, (PAST) may be linked to concepts of type SITUATION or its subtypes, such as STATE, EVENT, or ACT. The SITUATION occurs at a point in time, which has a successor, which is a time named Now. The name Now, like all names, must be unique within a context, but different contexts may have different instances of time that are named Now. For a database system, the graph in Fig. 3.16 might represent the sentence *A part number is a characteristic of a set of items, and* relation QOH(x, y) is

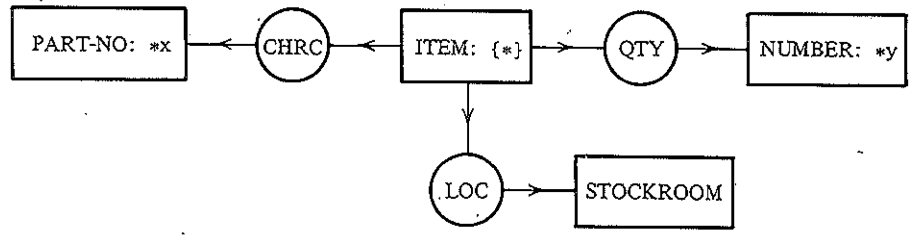

Fig. 3.16 Relational definition for quantity on hand (QOH)

a number is the quantity of such items located in a stock room. The body of the definition is a graph called the *relator* for the new conceptual relation labeled QOH, *quantity on hand*. The two formal parameters of QOH are marked *x and *y. The symbol {*}, which will be treated in Section 3.7, represents a generic set of individuals of type ITEM.

3.6.12 Assumption. A *relational definition*, written **relation** $t(a_1,...,a_n)$ **is** u, declares that the type label t for a conceptual relation is defined by the n-adic abstraction, $\lambda a_1,...,a_n u$. The body u is called the *relator* of t. If r is a conceptual relation of type t, the following conditions must be true:

- r has n arcs.
- If c_i is a concept linked to arc i of r, $type(c_i) \le type(a_i)$.

In an Aristotelian type hierarchy, all relational definitions could be reduced to a single dyadic relation type, LINK. Even linguistic relations like (AGNT) could be defined in terms of a concept of type AGENT:

relation `AGNT(x,y)` **is** `[ACT:*x]←(LINK)←[AGENT]→(LINK)→[ANIMATE:*y]`

where the type labels ANIMATE, AGENT, and ACT are themselves defined in terms of more primitive types. Since the real world is not Aristotelian, however, most types are specified in advance by a built-in set of standard types. The catalog of types listed in Appendix B is a sample starting set.

3.7.13 Assumption. In an Aristotelian type hierarchy T, there is a type label LINK for a dyadic conceptual relation. If t is a type label for a conceptual relation and $t \ne$LINK, then there exists a definition, **relation** $t(a_1,...,a_n)$ **is** u.

Relational contraction and expansion are the counterparts of type contraction and expansion. The relator of Fig. 3.16 may be contracted to

`[PART-NO]→(QOH)→[NUMBER],`

which in turn may be expanded back to the full graph form.

3.7.14 Assumption. The operation of *relational contraction* replaces a subgraph v of a conceptual graph w with a single conceptual relation r and the concepts linked to its arcs. Let $b_1,...,b_n$ be n distinct concepts of v, let v have no arcs linked to concepts in w-v, and let u be a copy of v with the concepts $b_1,...,b_n$ replaced by generic concepts $a_1,...,a_n$ where each b_i is a subtype of a_i. Then relational contraction consists of the following steps:

- Delete all of v from w except for $b_1,...,b_n$.
- Let $type(r)=\lambda a_1,...,a_n\ u$.
- For each i, link arc i of r to concept b_i.

If relational contraction is performed on a canonical graph, the resulting graph is canonical.

3.7.15 Definition. The operation of *relational expansion* replaces a conceptual relation and its attached concepts with the relator of a relational definition. Let w be a conceptual graph containing a conceptual relation r where $type(r)=\lambda a_1,...,a_n\ u$. Then relational expansion consists of the following steps:

- Detach r and its arcs from w.
- For each i, if b_i is the concept that was linked to arc i of r, then restrict a_i to $type(b_i)$. This restriction must be possible because of Assumption 3.6.12.
- For each i, join the restricted form of a_i to b_i.

3.7 AGGREGATION AND INDIVIDUATION

Every plural noun represents a set. In philosophy, Nelson Goodman (1951) populated the world with individuals that have other individuals as parts. For database semantics, Smith and Smith (1977) introduced *aggregation* as a method of defining composite entities with other entities as components. Mathematicians define sets that have other things, possibly other sets, as elements. Programming languages define structures that have data elements and other structures as components. The human body has parts like head, hands, and feet. Books have chapters, chapters have sections, sections have paragraphs, paragraphs have sentences, sentences have words, words have letters, and letters have strokes. Clubs have members, gaggles have geese, and even situations, events, and states may have subevents and aspects.

When the referents of concepts are limited to single individuals, as in Assumption 3.3.1, conceptual graphs cannot go beyond first-order logic. Yet many sentences in English cannot be expressed in first-order logic without a special notation for sets. Barwise and Cooper (1981) cited the following sentences, none of which can be translated into a first-order logic without set operations:

```
There are only a finite number of stars.
No one's heart will beat an infinite number of times.
More than half of John's arrows hit the target.
Most people voted for Carter.
```

To represent sets, a concept may contain a set of names or individual markers enclosed in braces: [DOG: {Snoopy,Lassie,Rin-Tin-Tin}]. The referent of this concept is a set of three individuals, all of whom conform to type DOG. When a concept has a set as referent, every element of the set must conform to the type label of the concept. To represent a set of mixed types such as cats, dogs, and lizards, but not palm trees, the type label ANIMAL or VERTEBRATE would be sufficiently general. To represent a completely arbitrary set, a concept could have the type label ⊤, which permits anything as referent.

Concepts with sets as referents are derived from ordinary concepts by *set join* and *set coercion*. As an example, the following two graphs may be joined on the concept [DANCE]:

```
[PERSON: Liz]←(AGNT)←[DANCE].
[PERSON: Kirby]←(AGNT)←[DANCE].
```

The result is a graph that contains two occurrences of the AGNT relation:

```
[PERSON: Liz]←(AGNT)←[DANCE]→(AGNT)→[PERSON: Kirby].
```

The two concepts of type PERSON may not be joined because they are individual concepts with conflicting referents. To make them joinable by set join, the first step is to perform a set coercion, which converts the concept [PERSON:Liz] to the concept [PERSON:{Liz}], which represents *a set of persons consisting of Liz*. After a similar set coercion on Kirby, the above graph becomes,

```
[PERSON: {Liz}]←(AGNT)←[DANCE]→(AGNT)→[PERSON: {Kirby}].
```

A set join of the two concepts of type PERSON forms a set referent that is the union of the sets {Liz} and {Kirby}:

```
[PERSON: {Liz,Kirby}]←(AGNT)←[DANCE].
```

This graph may be read *Liz and Kirby are dancing*. Note that both Liz and Kirby conform to type PERSON. A derivation of set referents by set coercions and set joins guarantees that all elements of the set conform to the type label of the concept.

3.7.1 Assumption. The referent of a concept c may be a set, every element of which must conform to *type*(c). If c is an individual concept with referent i, the operation of *set coercion* changes the referent of c to the *singleton set* $\{i\}$.

Set coercion introduces sets. The individual referents Liz and Kirby are first coerced to the singleton sets {Liz} and {Kirby}. Then set join combines them to form the set {Liz,Kirby}. Repeated joins can build up arbitrarily large sets.

3.7.2 Assumption. Let a and b be two concepts of the same type whose referents are sets. Then a and b may be joined by the operation of *set join*: first perform a join on the concepts a and b; then change the referent of the resulting concept to the union of *referent*(a) with *referent*(b).

Plural noun phrases represent concepts with sets as referents. When the elements are named explicitly, as *Bob and Charlie*, they map to a concept with an ordinary set as referent: [MAN: {Bob,Charlie}]. But the word *men* by itself is a plural word that does not name specific individuals. For such plurals, the symbol {*} represents a *generic set*, whose elements are unspecified. A generic set is related to a set of individuals in the same way as the generic marker * is related to an individual marker. The sentence *Niurka sees two men* would be represented by the graph,

```
[PERSON:Niurka]←(AGNT)←[SEE]→(OBJ)→[MAN:{*}]→(QTY)→[NUMBER:2].
```

The relation (QTY) links a concept whose referent is a set to a number that specifies how many elements are in the set.

3.7.3 Assumption. The symbol {*} represents a *generic set* of zero or more elements, which may occur as the referent of a concept. Set unions with {*} obey the following rules:

- *Empty set.* $\{\} \cup \{*\} = \{*\}$.
- *Generic set.* $\{*\} \cup \{*\} = \{*\}$.
- *Set of individuals.* $\{i_1,...,i_n\} \cup \{*\} = \{i_1,...,i_n,*\}$.

The set $\{i_1,...,i_n,*\}$ is called a *partially specified set*, which consists of the elements $i_1,...,i_n$ plus some unspecified others.

Just as measure contraction and name contraction were used in Section 3.3 to simplify conceptual graphs, the operation of *quantity contraction* may be used to simplify set referents. The symbol @ after a set shows that the following number represents the count of elements or *cardinality* of that set. With this notation, the sentence *Niurka sees two men* may be represented,

```
[PERSON:Niurka]←(AGNT)←[SEE]→(OBJ)→[MAN: {*}@2].
```

With a partially specified set as referent, the phrase *Norma, Frank, and two others* could be represented by the concept [PERSON: {Norma, Frank, *} @ 4].

Like ordinary joins, set joins generate graphs that are more specialized than the originals: they logically imply the graphs they are derived from, but they are not necessarily true if the originals are true. If Liz is dancing and Kirby is dancing, it is not necessarily true that they are dancing together. To handle such sentences in a query system, Dahl (1982) distinguished three different uses for plural noun phrases:

- *Collective.* All elements of a set participate in some relationship together: *Pat and her husband own the estate.*
- *Distributive.* Each element of a set satisfies some relation, but they do so separately: *Betty and Jerry are laughing.*
- *Respective.* Each element of an ordered sequence bears a particular relationship to a corresponding element of another sequence: *Dick, Jerry, Jimmy, and Ron are married to Pat, Betty, Rosalynn, and Nancy, respectively.*

In English, the semantics of the relation implicitly distinguishes the type of plural: two people cannot own the same thing separately, and two people cannot both laugh the same laugh. To be explicit, English uses the word *together* for the collective interpretation, *each* for the distributive, and *respectively* for the respective.

In conceptual graphs, the curly braces indicate a collective interpretation: {Albie,Clara,Ruby}. *Dist* in front of the set indicates a distributive interpretation: Dist{Albie,Clara,Ruby}. And *Resp* in front of a sequence delimited by angle brackets indicates a respective interpretation: Resp⟨Albie,Clara,Ruby⟩. As an example, consider the sentence *Two students read three books*, which could be represented as the graph,

```
[STUDENT: {*}@2]←(AGNT)←[READ]→(OBJ)→[BOOK: {*}@3].
```

This graph does not distinguish whether each student read every book, or one read two of them and the other read one. The following graph represents the sentence, *Two students each read three books.*

```
[STUDENT: Dist{*}@2]←(AGNT)←[READ]→(OBJ)→[BOOK: {*}@3].
```

The prefixes *Dist* and *Resp* indicate how the graphs containing set referents may be factored into graphs without set referents.

Another type of collection is the *disjunctive set*, a set of elements of which one participates in a relationship, but the particular one is not known. As an example, one way to represent the sentence *The elephant Clyde lives in either Africa or Asia* would use two separate propositions linked by an OR relation:

```
[PROPOSITION:
   [ELEPHANT:Clyde]→(STAT)→[LIVE]→(LOC)→[CONTINENT:Africa]]-
(OR)→[PROPOSITION:
   [ELEPHANT:Clyde]→(STAT)→[LIVE]→(LOC)→[CONTINENT:Asia]].
```

This graph, however, is a clumsy way of representing a simple sentence. A set join of the two parts would lead to the concept [CONTINENT:{Africa,Asia}], which would imply that Clyde lives in both places simultaneously. To show that only one of the elements is the actual referent, the elements of a disjunctive set are separated by the vertical bar "|" instead of commas:

```
[ELEPHANT:Clyde]→(STAT)→[LIVE]→(LOC)→[CONTINENT:{Africa|Asia}].
```

Disjunctive, collective, distributive, and respective sets are not four new kinds of sets. Rather, they are four different ways in which the elements of a standard set participate in a relationship.

3.7.4 Assumption. If the referent of a concept is a set, it may be one of four different kinds:

- A *collective set* in which the individuals in the set are separated by commas: $\{i_1,...,i_n\}$,
- A *disjunctive set* in which the individuals in the set are separated by vertical bars: $\{i_1 \mid ... \mid i_n\}$,
- A *distributive set* in which the individuals in the set are separated by commas, but the set itself is preceded by the prefix *Dist*: Dist$\{i_1,...,i_n\}$,
- A *respective set* in which the individuals are listed in a sequence delimited by angle brackets, and the set itself is preceded by the prefix *Resp*: Resp$\langle i_1,...,i_n \rangle$.

A set is a loose association between entities. There is no inherent connection between the elements other than the fact that they occur in the same collection. *Aggregation* is a tighter form of association: a *composite individual* is an aggregate of *components* that are linked by conceptual relations. The basis for an aggregation is some type definition, which sets up a pattern of concept and relation types:

```
type CIRCUS-ELEPHANT(x) is
   [ELEPHANT:*x]←(AGNT)←[PERFORM]→(LOC)→[CIRCUS].
```

A composite individual [CIRCUS-ELEPHANT:Jumbo] is defined by filling in the referent fields of generic concepts in the body of the type definition:

```
individual CIRCUS-ELEPHANT(Jumbo) is
   [ELEPHANT: Jumbo]←(AGNT)←[PERFORM: {*}]-
      (LOC)→[CIRCUS: Barnum & Bailey].
```

This graph defines *Jumbo* as the name of a circus elephant whose ELEPHANT component is Jumbo himself, whose PERFORM component is an unspecified set of performances, and whose CIRCUS component is named Barnum & Bailey. Smith and Smith (1977) introduced aggregation for grouping parts into a whole, and Brachman (1979) used the term *individuation* for specializing generic concepts to individual concepts. Both terms name aspects of the same process: aggregation groups individuals into a composite, and individuation projects a general graph into a composite of individuals.

In database terms, a *basis type* is a descriptor for a class of records, an aggregation is a record of particular data values, individuation is a mapping from the descriptor to the record, and a composite individual is a unique record identifier or *surrogate*. Brodie (1981) defined the entity type HOTEL-RESERVATION as the basis for an aggregation with components PERSON, ROOM, HOTEL, TIME-PERIOD, ARRIVAL-DATE, and DEPARTURE-DATE. In conceptual graphs, Brodie's type definition would take the following form:

```
type HOTEL-RESERVATION(reservation-no) is
   [RESERVATION: *reservation-no]-
      (RCPT)→ [PERSON]
      (OBJ)→  [ROOM]→(LOC)→[HOTEL]
      (DUR)→  [TIME-PERIOD]-
                 (STRT)→ [ARRIVAL-DATE]
                 (UNTL)→ [DEPARTURE-DATE].
```

The type definition explicitly shows the roles that each entity plays: a HOTEL-RESERVATION is a subtype of RESERVATION, it is identified by *reservation-no, and it is linked to a recipient PERSON, an object ROOM located in a HOTEL, and a duration TIME-PERIOD, which starts at an ARRIVAL-DATE and lasts until a DEPARTURE-DATE. A particular reservation identified by the marker #316209 is a composite individual:

```
individual HOTEL-RESERVATION(#316209) is
   [RESERVATION: #316209]-
      (RCPT)→ [PERSON: John Sowa]
      (OBJ)→  [ROOM: 2Q]→(LOC)→[HOTEL: Shelburne]
      (DUR)→  [TIME-PERIOD: @ 4 night]-
                 (STRT)→ [ARRIVAL-DATE: March 14, 1983]
                 (UNTL)→ [DEPARTURE-DATE: March 18, 1983].
```

Note the difference between a type and a composite individual: a type is specified by a differentia graph; a composite individual is specified by an aggregation in which one or more of the generic concepts of the differentia become individual concepts. For hotel reservation #316209, the aggregation contains exactly the same number of concepts and relations as the differentia, but in general, it may contain more. The projection that maps the differentia to the aggregation is called an *individuation*. Not all concepts in an aggregation need to have individual referents: generic concepts correspond to *null values* in the database. If the database contains many composite individuals of the same type, they may be stored in a compact form as *records* or *tuples* that contain only the referents and not the type labels and conceptual relations. The basis type and the record are sufficient to reconstruct the aggregation.

3.7.5 Definition. Let $t=\lambda a\ u$ be a type label; and let v be a canonical graph where $v \leq u$, π is a projection from u into v, and πa is an individual concept in v.

- The graph v is called an *aggregation* of *basis type* t.
- The projection π from the differentia u into the aggregation v is called an *individuation* of t.
- The individual i=*referent*(πa) is called a *composite individual*.
- For any concept c in u, *referent*(πc) is called the c *component* of the composite individual i.

Type definition associates a type label like CIRCUS-ELEPHANT with a graph of generic concepts. Aggregation associates a name like Jumbo or an individual marker like #316209 with a graph of individual concepts. In type expansion, the differentia for a type is joined to a concept. In *aggregate expansion*, the aggregation of a composite individual is joined to a concept. In either kind of expansion, implicit information in a name or type is made explicit.

3.7.6 Definition. Let u be a conceptual graph with a concept a in u where *referent*(a) is a composite individual with aggregation v and basis type *type*(a). Then *aggregate expansion* consists of joining the the concept a of u to the concept of v whose referent is the same as *referent*(a).

A type definition that includes concepts whose type labels are the same as the one being defined is said to be *directly recursive*. If the definition contains type labels that are supertypes of the one being defined, then it is *indirectly recursive*. The following type definition, which is both directly and indirectly recursive, defines a data structure similar to the list structures in the programming language LISP:

```
type LIST(x) is
   [DATA:*x]-
      (HEAD)→ [DATA]
      (TAIL)→ [LIST].
```

This definition says that a LIST is a type of DATA, which is linked via the relation (HEAD) to something of type DATA and via the relation (TAIL) to another LIST. The conceptual relations (HEAD) and (TAIL) have no primitive meaning in the theory of conceptual graphs, but their names were chosen to reflect their use in building list structures. By repeated type expansion of the concept [LIST], the following graph could be derived:

```
[LIST]-
   (HEAD)→ [DATA]
   (TAIL)→ [LIST]-
              (HEAD)→ [DATA]
              (TAIL)→ [LIST]-
                         (HEAD)→ [DATA]
                         (TAIL)→ [LIST].
```

Since LIST<DATA, the three concepts of type DATA could be restricted to LIST and then expanded to form the following graph. Note the placement of commas for closing the subtree linked to the most recent hyphen that is still open.

```
[LIST]-
   (HEAD)→ [LIST]-
              (HEAD)→ [DATA]
              (TAIL)→ [LIST],
   (TAIL)→ [LIST]-
              (HEAD)→ [LIST]-
                         (HEAD)→ [DATA]
                         (TAIL)→ [LIST],
              (TAIL)→ [LIST]-
                         (HEAD)→ [LIST]-
                                    (HEAD)→ [DATA]
                                    (TAIL)→ [LIST],
                         (TAIL)→ [LIST].
```

Such expansions could continue indefinitely. The expansion of type DATA could stop by restricting the type label to some type of data other than LIST. Expansion of type LIST could stop by reaching an individual of type LIST that could not be expanded further.

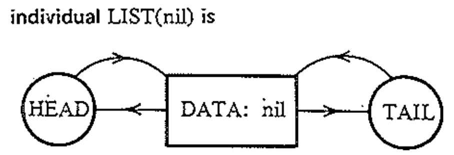

Fig. 3.17 A composite individual of type LIST

To define an individual that would stop the recursion, Fig. 3.17 introduces a composite individual of type LIST that is named *nil*. Because of the cycles in the aggregation, the indefinitely extended tree of type expansions would get mapped back onto itself once it reached a concept of the form [LIST:nil]. The convention that the head and tail of nil are nil itself is implemented in some versions of LISP.

EXERCISES

These exercises test the reader's understanding of the formalism. All readers should do Exercises 3.1, 3.2, and 3.3. Although some of the others require mathematical background, everyone should at least read them because they present interesting points that are not mentioned in the text.

3.1 Classify the anomalies in the following sentences as gibberish, ungrammatical, violations of category restrictions, violations of logic, or empirically false.

- Leon drew a square circle.
- People the the a book.
- All mimsy were the borogoves, and the mome raths outgrabe.
- Sincerity admires John.
- Nelson Rockefeller was elected president of the United States in 1968.

3.2 Translate each of the following conceptual graphs into an English sentence:

```
[PERSON: Clara]←(AGNT)←[WRITE]→(RSLT)→[PROGRAM].
[PERSON: Ivan]←(AGNT)←[THINK]-
   (OBJ)→[PROPOSITION: [PROGRAM:#]→(ATTR)→[GOOD]].
[GIVE]-
   (AGNT)→[PERSON: Ivan]
   (RCPT)→[PERSON: Clara]
   (OBJ)→ [PROMOTION].
```

3.3 Translate the following English sentences into conceptual graphs. Except for proper names, all the words, concepts, and conceptual relations needed for this exercise are listed in Appendix B.

- Dick shipped the hardware from New York to Cleveland via Buffalo.
- Jack is Charlie's philosophy teacher.
- Orders are received by telephone on Fridays.
- The cat is warm in the kitchen.
- Mark thinks that homework is difficult.

Note that some words in the lexicon, such as *the* and *is*, do not have corresponding concepts. Proper names appear in the referent field of a concept, not the type field.

3.4 Map the graphs drawn for the previous exercise into first-order logic according to the formula operator ϕ. Note that the sentence *Mark thinks that homework is difficult* cannot be mapped into a first-order formula by ϕ. Why not?

3.5 Choose three concept types, such as MOVE, HAPPY, and TOOL. Construct a hierarchy of subtypes for those types, using your own analysis as well as a dictionary, thesaurus, or synonym list.

3.6 Construct an example of a type hierarchy where some type labels do not have a unique maximal common subtype. Add new types to the hierarchy to convert it into a lattice.

3.7 Prove that if $s \leq t$, then $s = s \cap t$ and $t = s \cup t$. Since $t \leq t$, $t \cap t = t$ and $t \cup t = t$.

3.8 If two conceptual graphs u and v can be joined, then they must have a common specialization. If no concept of u can be joined to any concept of v, is it still possible for u and v to have a common specialization? When is it possible for two graphs u and v to have no common specializations. Give some examples.

3.9 Let v be a conceptual graph obtained by joining u and w on a compatible projection. If u and w happen to be the same graph, then v must always be smaller than u. But if u and w are different graphs, then v is usually larger than either u or w. In some cases, however, v may be smaller than either u or w. When is that possible? Give an example.

3.10 For a given set of canonical graphs C, there may be two or more different canonical bases from which all graphs in C can be derived. Give an example.

3.11 An earlier version of the theory (Sowa 1976) did not list simplification as a canonical formation rule. Instead, it gave the rule of *detachment*: if u is a canonical graph and r is any conceptual relation of u, then let the resulting graph w be any connected graph that remains when r and its arcs are detached from u.

- Prove that every canonical graph that is derivable by Assumption 3.4.3 is also derivable when the rule of simplification is replaced by the rule of detachment.
- Give some examples of derivations that are possible by detachment, but not by simplification.
- Prove that if every graph in a canonical basis has only a single conceptual relation, then Assumption 3.4.3 generates exactly the same canonical graphs as the rules that use detachment instead of simplification.
- If v was derived from u using the rule of detachment, show that u is not necessarily a generalization of v. (This is the reason why detachment was replaced in the current version of the formation rules.)

3.12 Two canonical graphs can be joined on a compatible projection that is not itself canonical, yet the result is canonical. Explain why the projection need not be canonical to make the result canonical.

3.13 The generalization hierarchy of conceptual graphs is not necessarily a tree or a lattice. In fact, even if the canon is finite, two finite canonical graphs could have an infinite number of common generalizations with no minimal element. Give some examples. (Hint: consider conceptual graphs that contain cycles.)

3.14 Let v be a common generalization of u_1 and u_2 with compatible projections $\pi_1 v$ in u_1 and $\pi_2 v$ in u_2. Prove that if $\pi_1 v$ and $\pi_2 v$ are not maximal compatible projections, then there must exist another graph v' that is also a common generalization of u_1 and u_2, v is a subgraph of v', and v' has exactly one more conceptual relation than v. Then v' is called a *local extension* of the common generalization v.

3.15 The following subranges are abbreviated abstractions. Expand each one into the corresponding λ-expression.

- $0 < \text{NUMBER} < 25$
- $\text{TEMP} > 90°\text{F}$
- $\text{MONEY} \geq \$100$

3.16 Prove the following statements:

- If v was derived from u by a type contraction, then ϕu is equivalent to ϕv.
- If v was derived from u by a minimal type expansion, then ϕu is equivalent to ϕv.
- If v was derived from u by a maximal type expansion, then ϕv implies ϕu.
- If v was derived from u by a relational contraction or a relational expansion, then ϕu is equivalent to ϕv.

3.17 In an Aristotelian type hierarchy with $\top$ as the only primitive concept type and LINK as the only primitive relation type, every other type must be defined in terms of some graph. What techniques can be used keep the differentia graphs distinct? Assume an ample supply of individual markers and give unique definitions for all the basic types.

3.18 Define a concept type LEGAL-CASE, where the definition includes concept types LAWYER, JUDGE, PROSECUTOR, DEFENDANT, and CHARGE. Then define an aggregate of basis type LEGAL-CASE where the components are individual concepts.

3.19 Readers who know LISP should map the list ((5 3) 7 9) into a conceptual graph that uses the type LIST and the individual nil defined in Section 3.7. This mapping illustrates list notation, but sets and aggregates are often more concise.

SUGGESTED READING

The earliest implementations of conceptual graphs were done for machine translation in the 1950s and 1960s; two of the most elaborate systems were Ceccato's *correlational nets* (1961, 1962) and Masterman's *semantic nets* (1961). Schank et al. (1975) developed conceptual dependency graphs to support natural language input, output, and reasoning. Fahlman (1979) proposed a direct hardware implementation for his NETL system. For a survey of various approaches, see the collection of papers edited by Findler (1979). Although many forms of these networks are used in AI, the philosophical and logical questions underlying them have often been ignored; Woods (1975), McDermott (1976), and Israel and Brachman (1981) analyze such issues and criticize the sloppy formulations of many theories in the field.

In this book, conceptual graphs are primarily applied to logic and language, but they could also be applied to spatial representations. Ballard and Brown (1982) is a good text on computer vision that covers general AI techniques as well; it includes about 150 pages of material on knowledge representation, semantic networks, graph matching, inference, and planning. For picture grammars, see Rosenfeld (1979).

For continuing studies on conceptual graphs and related representations, see the proceedings of various conferences on AI, computational linguistics, and cognitive science. Some important journals include *Artificial Intelligence*, *AI Magazine*, the *American Journal of Computational Linguistics*, and *Cognitive Science*. For the latest news, gossip, and reviews, see the SIGART *Newsletter*, published by the ACM Special Interest Group on Artificial Intelligence.

4 Reasoning and Computation

Conceptual graphs are a notation for representing knowledge. But to serve as a basis for thinking, they must be used in computation. This chapter introduces rules of inference for exact deduction, schemata for plausible reasoning, and actors for general computation.

4.1 SCHEMATA AND PROTOTYPES

In normal speech, people never say everything that can be said. Instead, they say just enough for the listener to reconstruct the intended meaning in the given context. According to Merleau-Ponty (1964), "The totality of meaning is never fully rendered: there is an immense mass of implications, even in the most explicit of languages; or rather, nothing is ever completely expressed, nothing exempts the subject who is listening from taking the initiative in giving an interpretation" (p. 29).

Because a human listener uses background knowledge to unravel the immense mass of implications, Bar-Hillel (1960) dismissed the possibility that a computer could ever do high quality machine translation: "What such a suggestion amounts to, if taken seriously, is the requirement that a translation machine should not only be supplied with a dictionary but also with a universal encyclopedia. This is surely utterly chimerical and hardly deserves any further discussion." Yet what seemed impossible with the small, slow, expensive machines of 1960 seems reasonable with the machines of the 1980s. Today, a single disk pack can store the *Encylopaedia Britannica* and transfer any page in a few milliseconds. But storage alone is not enough. Bar-Hillel made the further point that the machine must use the knowledge to make inferences, and "there exists so far no serious proposal for a scheme that would make a machine perform such inferences in the same or similar circumstances under which an intelligent human being would perform them."

Bar-Hillel correctly saw the need for storing and using background knowledge. But instead of looking for ways to represent that knowledge, he dismissed the prob-