Conceptual Graphs
For Representing Conceptual Structures

John F. Sowa

VivoMind Intelligence

Abstract. A conceptual graph (CG) is a graph representation for logic based on the semantic networks of artificial intelligence and the existential graphs of Charles Sanders Peirce. CG design principles emphasize the requirements for a cognitive representation: a smooth mapping to and from natural languages; an "iconic" structure for representing patterns of percepts in visual and tactile imagery; and cognitively realistic operations for perception, reasoning, and language understanding. The regularity and simplicity of the graph structures also support efficient algorithms for searching, pattern matching, and reasoning. Different subsets of conceptual graphs have different levels of expressive power: the ISO standard conceptual graphs express the full semantics of Common Logic (CL), which includes the subset used for the Semantic Web languages; a larger CG subset adds a context notation to support metalanguage and modality; and the research CGs are exploring an open-ended variety of extensions for aspects of natural language semantics. Unlike most notations for logic, CGs can be used with a continuous range of precision: at the formal end, they are equivalent to classical logic; but CGs can also be used in looser, less disciplined ways that can accommodate the vagueness and ambiguity of natural languages. This chapter surveys the history of conceptual graphs, their relationship to other knowledge representation languages, and their use in the design and implementation of intelligent systems.

1. Representing Conceptual Structures

Conceptual graphs are a notation for representing the conceptual structures that relate language to perception and action. Such structures must exist, but their properties can only be inferred from indirect evidence. Aristotle's inferences, with later extensions and qualifications, are still fundamentally sound: the meaning triangle of symbol, concept, and object; logic as a method of analyzing reasoning (*logos*); and a hierarchy of psyches ranging from the vegetative psyche of plants, the psyche of primitive animals like sponges, the locomotive pysche of worms, the imagery of psyches with sight and hearing, to the human psyche of an animal having logos (*zôon logon echein*). The medieval Scholastics extended Aristotle's logic, linguistics, and psychology, but Locke, Condillac, and the British empiricists developed more loosely structured theories about associations of ideas. Kant introduced schemata as tightly structured patterns of concepts and percepts, which became the foundation for many later developments. Peirce integrated aspects of all these proposals with modern logic in constructing a theory of signs that he called *semeiotic*.

In the 20th century, behaviorists tried to avoid hypotheses about conceptual structures; fortunately, many psychologists ignored them. Otto Selz (1913, 1922), who was dissatisfied with the undirected associationist theories, adapted Kant's schemata for a goal-directed theory he called *schematic anticipation*. Selz represented each schema as a network of concepts that contained empty slots, and he asked subjects to suggest appropriate concepts to fill the slots while he recorded their verbal protocols. Figure 1 shows a schema that Selz used in his experiments.

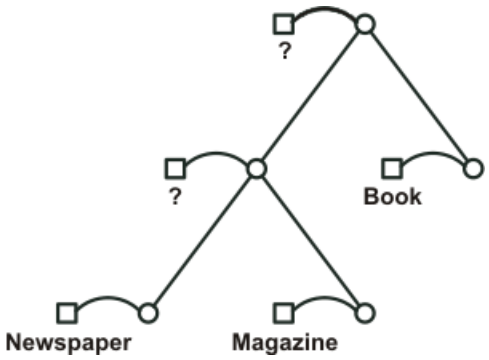

Figure 1. A schema used by Otto Selz

The expected answers to the question marks in Figure 1 are generalizations of the words at the bottom: the supertype of Newspaper and Magazine is Periodical, and the supertype of Periodical and Book is Publication. After analyzing the methods subjects use to solve such puzzles, Selz proposed a theory of goal-directed search that starts with a schema as an anticipation of the final result and propagates the question marks to subsidiary schemata. Selz's theories have a strong similarity to the backtracking methods developed by Newell and Simon (1972). That similarity is not an accident. Newell and Simon learned Selz's theories from one of their visitors, the psychologist Adriaan de Groot, who used Selz's methods to study the thinking processes of chessplayers. One of their students, Quillian (1968), cited Selz as a source for his version of semantic networks. For computation, Quillian designed a *marker passing* algorithm, inspired by Selz's ideas for propagating question marks from one schema to another.

Another source for semantic networks was the *dependency grammar* developed by Lucien Tesnière (1959). Figure 2 shows a dependency graph for the sentence *L'autre jour, au fond d'un vallon, un serpent piqua Jean Fréron* (The other day, at the bottom of a valley, a snake stung Jean Fréron). At the top is the verb *piqua* (stung); each word below it depends on the word above to which it is attached. The bull's eye symbol indicates an implicit preposition (*à*).

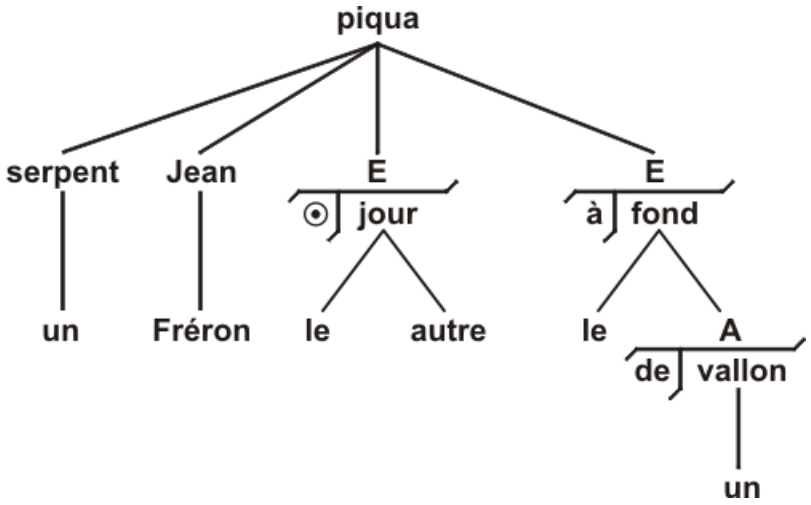

Figure 2. A dependency graph in Tesnière's notation

Tesnière had a major influence on linguistic theories that place more emphasis on semantics than syntax. Hays (1964) proposed dependency graphs as an alternative to Chomsky's notation, and Klein and Simmons (1963) developed a related version for machine translation. Those systems influenced Schank (1975), who adopted dependency graphs, but shifted the emphasis to concepts rather than words. Figure 3 shows a *conceptual dependency graph* for the sentence *A dog is greedily eating a bone*. Instead of Tesnière's tree notation, Schank used different kinds of arrows for different relations, such as

⇔ for the agent-act relation and an arrow marked with *o* for object or *d* for direction. He also replaced the words *eat* and *greedily* with labels that represent the concept types `Ingest` and `Greedy`. The subscript 1 on `Dog` indicates that the bone went into the same dog that ingested it.

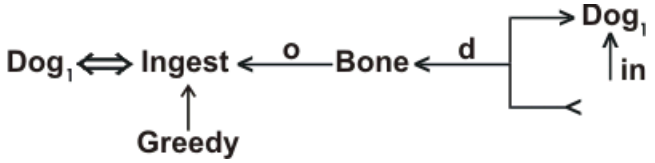

Figure 3. Schank's notation for conceptual dependencies

The early semantic networks were used for machine translation and question answering, but they could not represent all the features of logic. The first publication on conceptual graphs (Sowa 1976) combined semantic networks with the quantifiers of predicate calculus and labeled the links between concepts with the *case relations* or *thematic roles* of linguistics (Fillmore 1968). That paper also presented a graph grammar for conceptual graphs based on four *canonical formation rules*. As an application, it illustrated CGs for representing natural language questions and mapping them to *conceptual schemata*. Each schema contained a declarative CG with attached *actor nodes* that represented functions or database relations. For computation, it proposed two kinds of marker passing for invoking the actors: backward-chaining markers, as in the networks by Selz and Quillian, and forward-chaining markers, as in Petri nets (Petri 1965). As an example of the 1976 notation, Figure 4 shows a conceptual graph for the sentence *On Fridays, Bob drives his Chevy to St. Louis*.

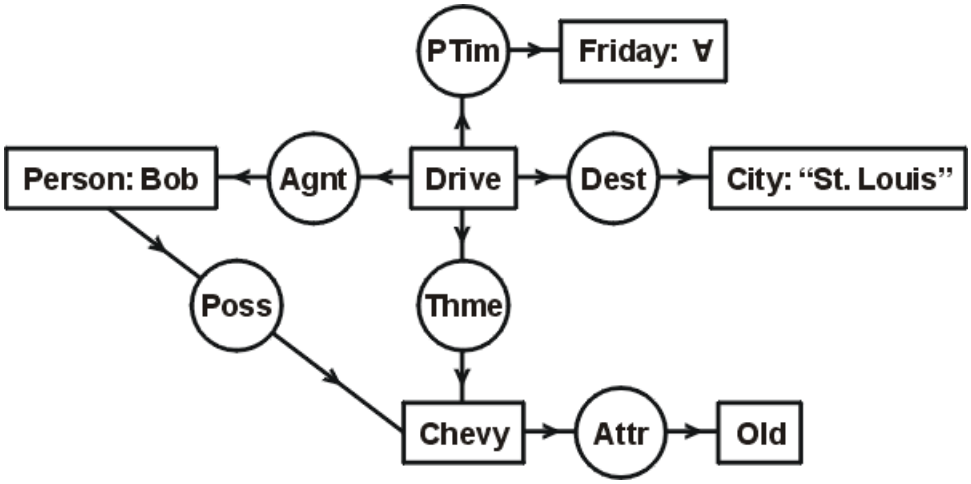

Figure 4. ***On Fridays, Bob drives his old Chevy to St. Louis.***

The rectangles in Figure 4 are called *concept nodes*, and the circles are called *conceptual relation nodes*. An arc pointing toward a circle marks the first *argument* of the relation, and an arc pointing away from a circle marks the last argument. If a relation has only one argument, the arrowhead is omitted. If a relation has more than two arguments, the arrowheads are replaced by integers 1,...,*n*. Each concept node has a *type label*, which represents the type of entity the concept refers to: **`Friday`**, **`Person`**, **`Drive`**, **`City`**, **`Chevy`**, or **`Old`**. One of the concepts has a *universal quantifier* ∀ to represent *every Friday*; two concepts identify their referents by the names **`Bob`** and **`"St. Louis"`**; and the remaining three concepts represent the existence of a Chevy, an instance of driving, and an instance of oldness. Each of the six relation nodes has a label that represents the type of relation: agent **`(Agnt)`**, point-in-time **`(PTim)`**, destination **`(Dest)`**, possession **`(Poss)`**, theme **`(Thme)`**, or attribute **`(Attr)`**. The CG as a whole asserts that on every Friday, the person Bob, who possesses an old Chevy, drives it to St. Louis. Figure 4 can be translated to the following formula in a typed version

of predicate calculus:

```
(∀x1:Friday)(∃x2:Drive)(∃x3:Chevy)(∃x4:Old)
(Person(Bob) ∧ City("St. Louis") ∧ PTim(x2,X1)
   ∧ Agnt(x2,Bob) ∧ Poss(Bob,x3) ∧ Thme(x2,x3)
   ∧ Attr(x3,x4) ∧ Dest(x2,"St. Louis"))
```

As this translation shows, any concept without a name or a universal quantifier has an implicit existential quantifier. The default assumption for scope gives the universal quantifier higher precedence than existentials. That leaves open the question whether Bob drives the same old Chevy or a different one on each Friday.

A later version of CGs (Sowa 1984) used Peirce's *existential graphs* (EGs) as the logical foundation. An important feature of EGs is an explicit enclosure to delimit the scope of quantifiers and other logical operators. The CG in Figure 5 has a large *context box* as a delimiter; the subgraph for Bob and his old Chevy is outside that scope. Since the referent for the city of St. Louis is designated by a proper name, it is a constant, which can be left inside the box or moved outside the box without any change to the logic. The resulting CG represents *Bob has an old Chevy, and on Fridays, he drives it to St. Louis.*

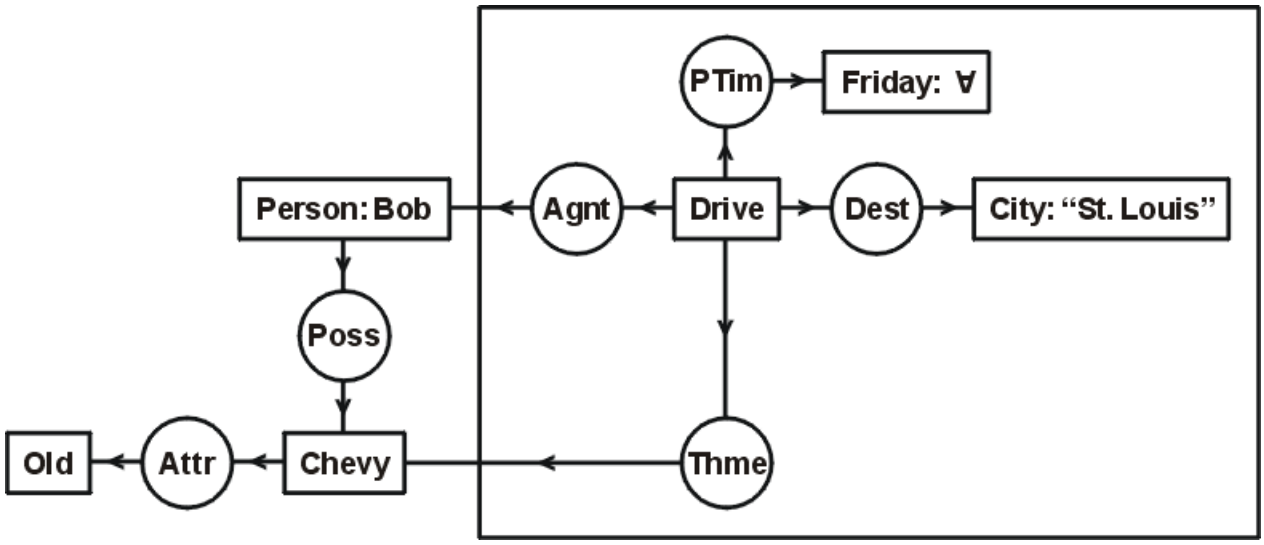

Figure 5. A context box for delimiting the scope of quantifiers

The translation of Figure 5 to predicate calculus moves the quantifiers for Bob and his old Chevy in front of the universal. The only existential quantifier that remains within the scope of the universal is the one for the concept **[Drive]**:

```
(∃x1:Chevy)(∃x2:Old)(Person(Bob) ∧ Poss(Bob,x1) ∧ Attr(x1,x2)
   ∧ (∀x3:Friday)(∃x4:Drive)(City("St. Louis") ∧ PTim(x4,X3)
      ∧ Agnt(x4,Bob) ∧ Thme(x4,x1) ∧ Dest(x4,"St. Louis")))
```

The graphical notation illustrated in Figures 4 and 5 is called the CG *display form*. A linear representation for CGs, called the *Conceptual Graph Interchange Format* (CGIF), is one of the three standard dialects for *Common Logic* (ISO/IEC 24707). CGIF has a one-to-one mapping to and from the nodes of the display form. Following is the CGIF version of Figure 5:

```
[Person Bob] [Chevy *x1] [Old *x2] (Poss Bob ?x1) (Attr ?x1 ?x2)
[ [Friday @every*x3] [Drive *x4] [City "St. Louis"] (PTim ?x4 ?x3)
  (Agnt ?x4 Bob) (Thme ?x4 ?x1) (Dest ?x2 "St. Louis") ]
```

Square brackets represent the concept nodes, and parentheses represent the relation nodes. Connections between concepts and relations are indicated by names, such as **Bob** and **"St. Louis"** or by *coreference labels* such as ***x1** and **?x1**. Special characters such as ∀ are represented by ASCII strings such as **@every**; the conjunction symbol ∧ is not needed, since the graph implies a conjunction of all the nodes in the same context. The first occurrence of any coreference label, called a *defining label*, is marked by an asterisk ***x1**; *bound labels*, marked by a question mark **?x1**, indicate a link to the node that contains the corresponding defining label. Note that **Friday** and **Chevy** represent types, not names of instances. Type labels are placed on the left side of a node; names, quantifiers, and coreference labels are on the right.

Graphs have advantages over linear notations in human factors and computational efficiency. As Figures 4 and 5 illustrate, the CG display form can show relationships at a glance that are harder to see in the linear notations for logic. Graphs also have a highly regular structure that can simplify many algorithms for searching, pattern matching, and reasoning. Although nobody knows how any information is represented in the brain, graphs minimize extraneous detail: they show connections directly, and they avoid the ordering implicit in strings or trees. The remaining sections of this chapter develop these ideas further: natural logic in Section 2; reasoning methods in Section 3; context and metalanguage in Section 4; research issues in Section 5; and the ISO standard for Common Logic in the appendix, which includes the CGIF grammar.

2. Toward a Natural Logic

From Aristotle and Euclid to Boole and Peirce, the twin goals of logic were to understand the reasoning processes in language and thought and to develop a tool that could facilitate reasoning in philosophy, mathematics, science, and practical applications. Various researchers put more emphasis on one goal or the other. Selz, Tesnière, and many linguists and psychologists studied the mechanisms underlying language and thought. Frege and most 20th-century logicians emphasized the applications to mathematics. Peirce and some AI researchers put equal emphasis on both. During his long career, Peirce invented several different notations for logic: an algebra of relations (1870) that is similar to the relational algebra used in database systems; an extension to Boolean algebra (1880, 1885), which became the modern predicate calculus; and existential graphs (1906, 1909), whose inference rules, he claimed, present "a moving picture of thought."

The first complete version of first-order logic was a tree notation by Frege (1879) called *Begriffsschrift* (concept writing). For his trees, Frege used only four operators: assertion (the "turnstile" operator ⊢), negation (a short vertical line), implication (a hook), and the universal quantifier (a cup containing the bound variable). Figure 6 shows Frege's notation for the sentence *Bob drives his Chevy to St. Louis*.

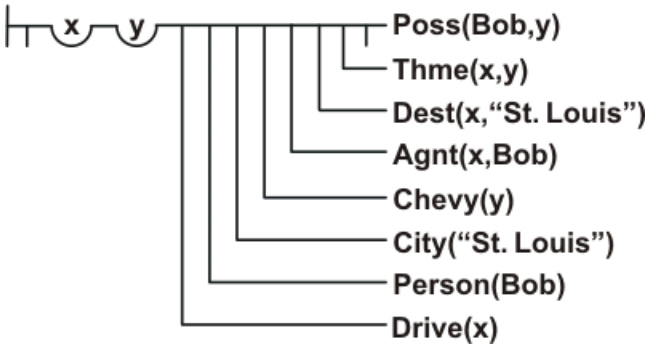

Figure 6. Frege's Begriffsschrift for *Bob drives his Chevy to St. Louis.*

Frege had a contempt for language, and set out "to break the domination of the word over the human spirit by laying bare the misconceptions that through the use of language often almost unavoidably arise concerning the relations between concepts." His choice of operators simplified his rules of inference, but they led to cumbersome paraphrases in natural language. A direct translation of Figure 6 to predicate calculus would be

~(∀*x*)(∀*y*) (Drive(*x*) ⊃ (Person(Bob) ⊃ (City("St. Louis") ⊃ (Chevy(*y*) ⊃
(Agnt(*x*,Bob) ⊃ (Dest(*x*,"St. Louis") ⊃ (Thme(*x*,*y*) ⊃ ~Poss(Bob,*y*))))))))

In English, this formula could be read *It is false that for every x and y, if x is an instance of driving then if Bob is a person then if St. Louis is a city then if y is a Chevy then if the agent of x is Bob then if the destination of x is St. Louis then if the theme of x is y then Bob does not possess y.*

Although Peirce had invented the algebraic notation for predicate calculus, he believed that a graph representation would be more cognitively realistic. While he was still developing the algebra of logic, he experimented with a notation for *relational graphs*. Figure 7 shows a relational graph for the same sentence as Figure 6. In that graph, an existential quantifier is represented by line or by a *ligature* of connected lines, and conjunction is the default Boolean operator. Since those graphs did not represent proper names, monadic predicates **`isBob`** and **`isStLouis`** are used to represent names.

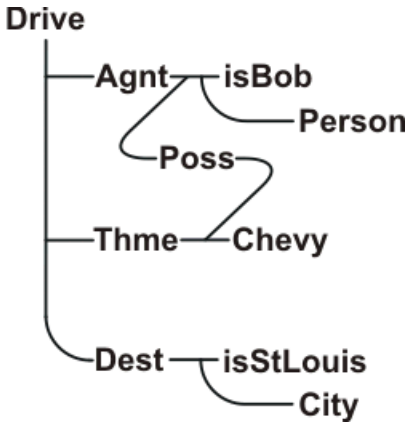

Figure 7. A relational graph for *Bob drives a Chevy to St. Louis.*

Figure 7 contains four ligatures: one for the instance of driving, one for Bob, one for the Chevy, and one for St. Louis. Each ligature maps to an existentially quantified variable in predicate calculus:

(∃*x*)(∃*y*)(∃*z*)(∃*w*)(Drive(*x*) ∧ Agnt(*x*,*y*) ∧ Person(*y*) ∧ isBob(*y*) ∧ Poss(*y*,*z*) ∧
Thme(*x*,*z*) ∧ Chevy(*z*) ∧ Dest(*x*,*w*) ∧ City(*w*) ∧ isStLouis(*w*))

Peirce experimented with various graphic methods for representing the other operators of his algebraic notation, but like the AI researchers of the 1960s, he couldn't find a good way to express the scope of quantifiers and negation. In 1897, however, he discovered a simple, but brilliant innovation for his new version of *existential graphs* (EGs): an oval enclosure for showing scope. The default operator for an oval with no other marking is negation, but any metalevel relation can be linked to the oval. The graph on the left of Figure 8 is an EG for the sentence *If a farmer owns a donkey, then he beats it*. Since an implication $p \supset q$ is equivalent to $\sim(p \wedge \sim q)$, the nest of two ovals expresses **if** *p* **then** *q*. To enhance the contrast, Peirce would shade any area enclosed in an odd number of negations.

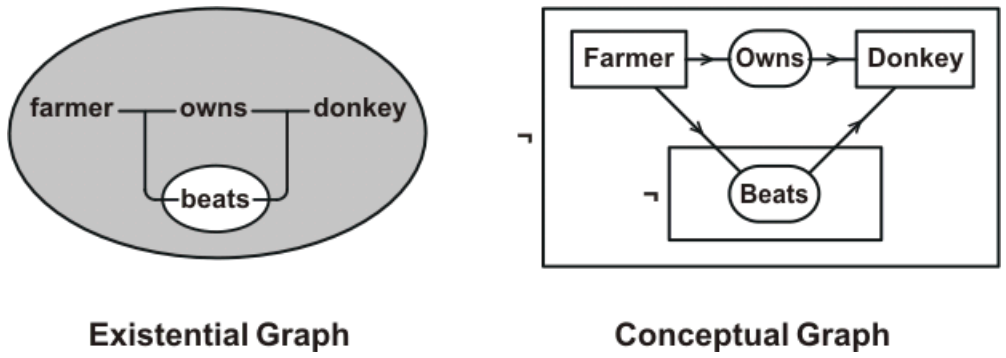

Figure 8. EG and CG for *If a farmer owns a donkey, then he beats it.*

The equivalent conceptual graph is on the right of Figure 8. Since boxes nest better than ovals, Peirce's ovals are represented by rectangles marked with the symbol ¬ for negation. The choice of ovals or boxes, however, is a trivial difference. Three other differences are more significant: first, each rectangle can be interpreted as a concept node to which conceptual relations other than negation may be attached; second, the existential quantifiers, which are represented by EG lines, are represented by CG nodes, which may contain proper names, universal quantifiers, or even *generalized quantifiers*; and third, the type labels on the left side of a concept node restrict the range of quantifiers. Therefore, the EG maps to an untyped formula:

~(∃*x*)(∃*y*)(Farmer(*x*) ∧ Donkey(*y*) ∧ Owns(*x*,*y*) ∧ ~Beats(*x*,*y*))

But the CG maps to the logically equivalent typed formula:

~(∃*x*:Farmer)(∃*y*:Donkey)(Owns(*x*,*y*) ∧ ~Beats(*x*,*y*))

In order to preserve the correct scope of quantifiers, the implication operator ⊃ cannot be used to represent an English *if-then* sentence unless the existential quantifiers are converted to universals and moved to the front:

(∀*x*)(∀*y*)((Farmer(*x*) ∧ Donkey(*y*) ∧ Owns(*x*,*y*)) ⊃ Beats(*x*,*y*))

In English, this formula could be read *For every x and y, if x is a farmer who owns a donkey y, then x beats y.* The unusual nature of this paraphrase led Kamp (1981) to develop *discourse representation structures* (DRSs) whose logical structure is isomorphic to Peirce's existential graphs (Figure 9).

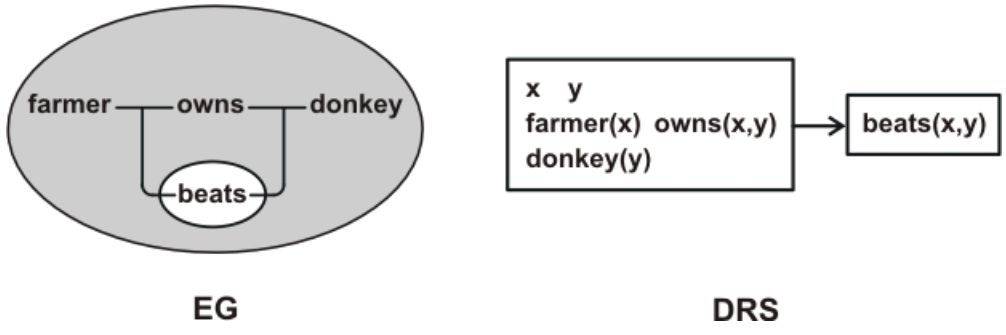

Figure 9. EG and DRS for *If a farmer owns a donkey, then he beats it.*

Kamp's primitives are the same as Peirce's: the default quantifier is the existential, and the default Boolean operator is conjunction. Negation is represented by a box, and implication is represented by two boxes. As Figure 9 illustrates, the EG contexts allow quantifiers in the *if* clause to include the *then*

clause in their scope. Although Kamp connected his boxes with an arrow, he made the same assumption about the scope of quantifiers. Kamp and Reyle (1993) went much further than Peirce in analyzing discourse and formulating the rules for interpreting anaphoric references, but any rule stated in terms of the DRS notation can also be applied to the EG or CG notation.

The CG in Figure 8 represents the verbs *owns* and *beats* as dyadic relations. That was the choice of relations selected by Kamp, and it can also be used with the EG or CG notation. Peirce, however, noted that the event or state expressed by a verb is also an entity that can be referenced by a quantified variable. That point was independently rediscovered by linguists, computational linguists, and philosophers such as Davidson (1967). The CG in Figure 10 represents events and states as entities linked to their participants by linguistic relations. The type labels `If` and `Then` indicate negated contexts; the two new relation types are `Expr` for experiencer and `Ptnt` for patient.

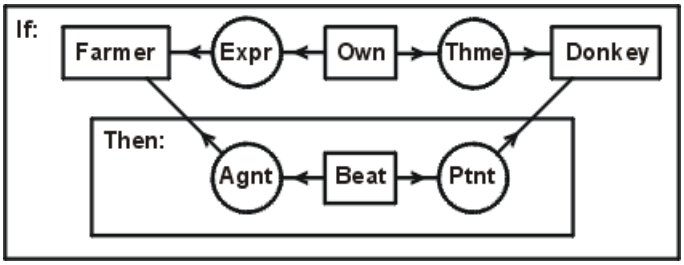

Figure 10. CG with case relations shown explicitly

All the notations in this section, including the diagrams and predicate calculus, can express full first-order logic. Since natural languages can also express full FOL, that expressive power is a requirement for a natural logic. But natural languages can express much more or much less, and they can be vague, precise, cryptic, erudite, or naive. Therefore, a natural logic must be flexible enough to vary the level of expression and precision as needed. The choice of logical operators is one consideration. Frege's choice is awkward for the sample sentence, and Peirce's choice is better. Predicate calculus has five or six common operators, and it can accommodate new ones as needed. For mathematics, many choices of primitives, such as Frege's or Peirce's, can be used to define all the others. For empirical subjects, however, conjunction and the existential quantifier are the only operators that can be observed directly, and the others must be inferred from indirect evidence. Therefore, Peirce's choice of primitives combined with a mechanism for defining other operators seems appropriate.

The structure of logical expressions is another consideration. As Peirce, Tesnière, and the AI researchers have shown, graphs can be directly mapped to natural languages, they can be as precise as any other notation for logic, and they can represent vague sentences as easily as precise ones. Graphs have the minimum syntax needed to show connections, but something more is required to show the scope of the logical operators. The fact that Peirce and Kamp independently discovered isomorphic enclosures for showing scope is an important point in their favor. Those arguments suggest that relational graphs with the Peirce-Kamp enclosures are a good candidate for the structure. The CG extensions show that more features can be added while preserving the overall simplicity. A truly natural logic should support reasoning methods that could also be considered natural.

3. A Moving Picture of Thought

Like the graphs themselves, Peirce's rules for graph logics are simple enough to be considered candidates for a natural logic: each rule of inference inserts or erases a single graph or subgraph. Peirce's proof procedure is a generalization and simplification of *natural deduction*, which Gentzen

(1935) invented 30 years later. Although Peirce originally stated his rules in terms of the EG syntax, they can be restated in a form that depends only on the semantics. That approach has a striking similarity to recent work on the category-theoretic treatment of proofs (Hughes 2006; McKinley 2006). As a psychological hypothesis, a syntax-independent procedure can be evaluated in terms of external evidence without any assumptions about internal representations.

To implement a proof procedure, semantic rules must be related to syntax. In 1976, the canonical formation rules were presented as a generative grammar for CGs. The 1984 version was also a generative grammar, but the rules were explicitly related to their semantic effects. The latest version (Sowa 2000) classifies the rules in three groups according to their semantic effects: *equivalence*, *specialization*, and *generalization*. Each rule transforms a starting graph or graphs *u* to a resulting graph *v*:

- **Equivalence**. *Copy* and *simplify* are equivalence rules, which generate a graph *v* that is logically equivalent to the original: $u \supset v$ and $v \supset u$. Equivalent graphs are true in exactly the same models.
- **Specialization**. *Join* and *restrict* are specialization rules, which generate a graph *v* that implies the original: $v \supset u$. Specialization rules monotonically decrease the set of models in which the result is true.
- **Generalization**. *Detach* and *unrestrict* are generalization rules, which generate a graph *v* that is implied by the original: $u \supset v$. Generalization rules monotonically increase the set of models in which the result is true.

Each rule has an inverse rule that undoes any change caused by the other. The inverse of copy is simplify, the inverse of restrict is unrestrict, and the inverse of join is detach. Combinations of these rules, called *projection* and *maximal join*, perform larger semantic operations, such as answering a question or comparing the relevance of different alternatives. The next three diagrams (Figures 11, 12, and 13) illustrate these rules with *simple graphs*, which use only conjunction and existential quantifiers.

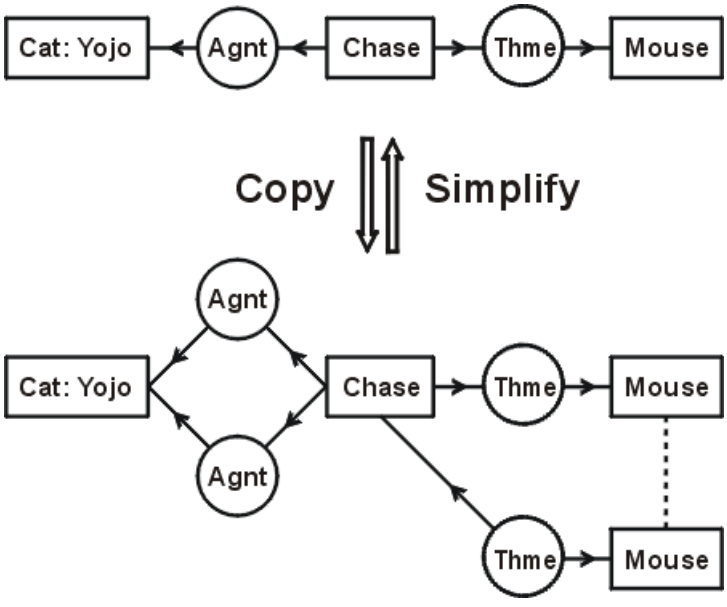

Figure 11. Copy and simplify rules

The CG at the top of Figure 11 represents the sentence *The cat Yojo is chasing a mouse*. The down arrow represents two applications of the copy rule. One application copies the **(Agnt)** relation, and a

second copies the subgraph →**(Thme)**→**[Mouse]**. The dotted line connecting the two **[Mouse]** concepts is a *coreference link*, which indicates that both concepts refer to the same individual; it represents equality in predicate calculus. The up arrow represents two applications of the simplify rule, which performs the inverse operations of erasing redundant copies. Following are the CGIF sentences for both graphs:

```
[Cat: Yojo] [Chase: *x] [Mouse: *y] (Agent ?x Yojo) (Thme ?x ?y)

[Cat: Yojo] [Chase: *x] [Mouse: *y] [Mouse: ?y]
(Agent ?x Yojo) (Agent ?x Yojo) (Thme ?x ?y) (Thme ?x ?y)
```

As the CGIF illustrates, the copy rule makes redundant copies, which are erased by the simplify rule. In effect, the copy rule is $p \supset (p \wedge p)$, and the simplify rule is $(p \wedge p) \supset p$.

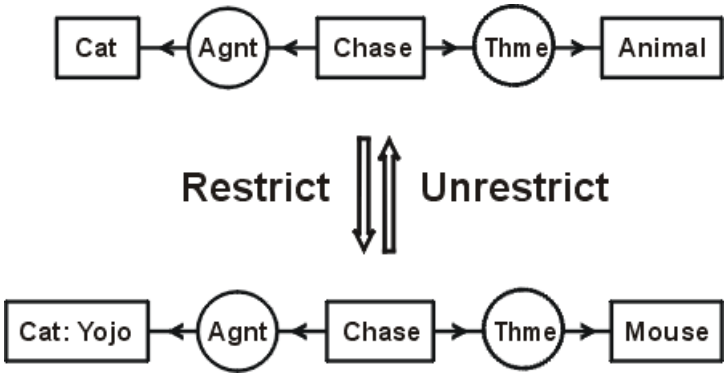

Figure 12. Restrict and unrestrict rules

The CG at the top of Figure 12 represents the sentence *A cat is chasing an animal*. By two applications of the restrict rule, it is transformed to the CG for *The cat Yojo is chasing a mouse*. In the first step, the concept **[Cat]**, which says that there exists some cat, is *restricted by referent* to the more specific concept **[Cat: Yojo]**, which says that there exists a cat named Yojo. In the second step, the concept **[Animal]**, which says that there exists an animal, is *restricted by type* to a concept of a subtype **[Mouse]**. The more specialized graph implies the more general one: if the cat Yojo is chasing a mouse, then a cat is chasing an animal. To show that the bottom graph v implies the top graph u, let c be a concept of u that is being restricted to a more specialized concept d, and let u be $c \wedge w$, where w is the remaining information in u. By hypothesis, $d \supset c$. Therefore, $(d \wedge w) \supset (c \wedge w)$. Hence, $v \supset u$.

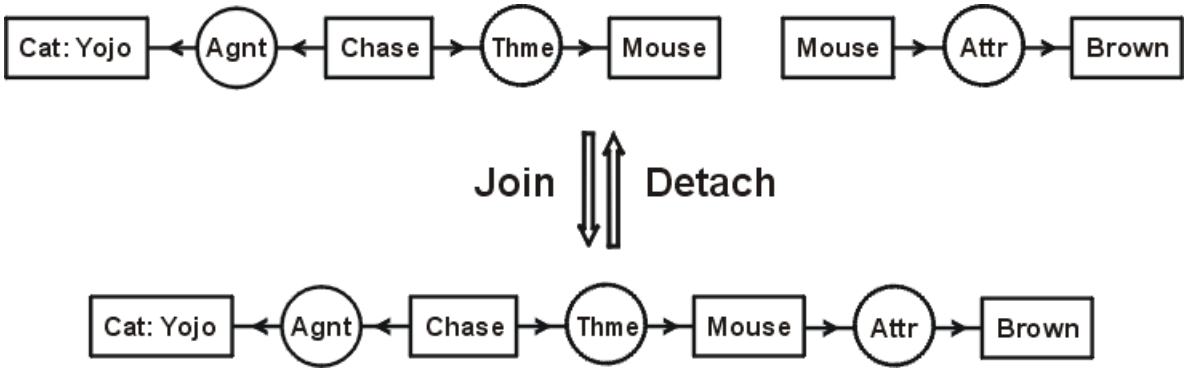

Figure 13. Join and detach rules

At the top of Figure 13 are two CGs for the sentences *Yojo is chasing a mouse* and *A mouse is brown*. The join rule overlays the two identical copies of the concept **[Mouse]** to form a single CG for the

sentence *Yojo is chasing a brown mouse*. The detach rule undoes the join to restore the top graphs. Following are the CGIF sentences that represent the top and bottom graphs of Figure 13:

```
[Cat: Yojo] [Chase: *x] [Mouse: *y] (Agent ?x Yojo) (Thme ?x ?y)
[Mouse: *z] [Brown: *w] (Attr ?z ?w)

[Cat: Yojo] [Chase: *x] [Mouse: *y] (Agent ?x Yojo) (Thme ?x ?y)
[Brown: *w] (Attr ?y ?w)
```

As the CGIF illustrates, the bottom graph consists of substituting y for every occurrence of z in the top graph and erasing redundant copies. In general, every join assumes an equality of the form $y=z$ and simplifies the result. If q is the equality and u is the top pair of graphs, then the bottom graph is equivalent to $q \wedge u$, which implies u. Therefore, the result of join implies the original graphs. Together, the generalization and equivalence rules are sufficient for a complete proof procedure for the simple graphs with no negations. The specialization and equivalence rules can be used in a refutation procedure for a proof by contradiction.

To handle full first-order logic, rules for negations must be added to the canonical formation rules. Peirce defined a complete proof procedure for FOL whose rules depend on whether a context is positive (nested in an even number of negations, possibly zero) or negative (nested in an odd number of negations). Those rules are grouped in three pairs: one rule (i) inserts a graph, and the other (e) erases a graph. The only axiom is a blank sheet of paper (an empty graph with no nodes). In effect, the blank is a generalization of all other graphs. Following is a restatement of Peirce's rules in terms of specialization and generalization. These same rules apply to both propositional logic and full first-order logic. In FOL, the operation of inserting a coreference link between two nodes has the effect of identifying them (i.e., inserting an equality); erasing a coreference link has the inverse effect of erasing an equality. In a linear notation, such as CGIF or predicate calculus, the operation of inserting or erasing an equality requires an additional operation of renaming labels. In a pure graph notation, there are no labels and no need for renaming.

1. (i) In a negative context, any graph or subgraph (including the blank) may be replaced by any specialization.

   (e) In a positive context, any graph or subgraph may be replaced by any generalization (including the blank).

2. (i) Any graph or subgraph in any context *c* may be copied in the same context *c* or into any context nested in *c*. (No graph may be copied directly into itself. But it is permissible to copy a graph *g* in the context *c* and then make a copy of the copy inside the original *g*.)

   (e) Any graph or subgraph that could have been derived by rule 2i may be erased. (Whether or not the graph was in fact derived by 2i is irrelevant.)

3. (i) A double negation (nest of two negations with nothing between the inner and outer) may be drawn around any graph, subgraph, or set of graphs in any context.

   (e) Any double negation in any context may be erased.

This version of the rules was adapted from a tutorial on existential graphs by Peirce (1909). When these rules are applied to CGIF, some adjustments may be needed to rename coreference labels or to convert a bound label to a defining label or vice versa. For example, if a defining node is erased, some bound label **?x** may become the new defining label ***x**. Such adjustments are not needed in the pure graph notation.

All the axioms and rules of inference for classical FOL, including the rules of the *Principia Mathematica*, natural deduction, and resolution, can be proved in terms of Peirce's rules. As an example, Frege's first axiom, written in the algebraic notation, is ***a*⊃(*b*⊃*a*)**. Figure 14 shows a proof by Peirce's rules.

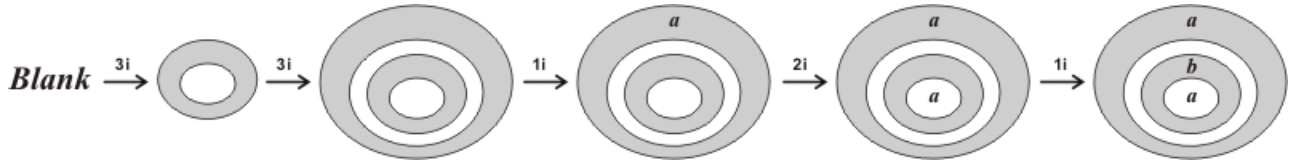

Figure 14. Proof of Frege's first axiom by Peirce's rules

In CGIF, the propositions *a* and *b* can be represented as relations with zero arguments: **(a)** and **(b)**. Following are the five steps of Figure 14:

1. By rule 3i, Insert a double negation around the blank: **~[~[ ]]**
2. By 3i, insert a double negation around the previous one: **~[~[~[~[ ]]]]**
3. By 1i, insert (a): **~[(a) ~[~[~[ ]]]]**
4. By 2i, copy (a): **~[(a) ~[~[~[(a)]]]]**
5. By 1i, insert (b): **~[(a) ~[~[(b) ~[(a)]]]]**

The theorem to be proved contains five symbols, and each step of the proof inserts one symbol into its proper place in the final result. All the axioms of any version of FOL could be derived from a blank by similarly short proofs.

Frege's two rules of inference were *modus ponens* and *universal instantiation*. Figure 15 is a proof of *modus ponens*, which derives *q* from a statement *p* and an implication ***p*⊃*q***:

Figure 15. Proof of modus ponens

Following is the CGIF version of Figure 15:

1. Starting graphs: **(p) ~[(p) ~[(q)]]**
2. By 2e, erase the nested copy of **(p)**: **(p) ~[~[(q)]]**
3. By 1e, erase (p): **~[~[(q)]]**
4. By 3e, erase the double negation: **(q)**

The rule of *universal instantiation* allows any term *t* to be substituted for a universally quantified variable in a statement of the form **(∀*x*)P(*x*)** to derive **P(*t*)**. In EGs, the term *t* would be represented by a graph of the form **–*t***, which states that something satisfying the condition *t* exists: **(∃*y*)*t*(*y*)**. The universal quantifier ∀ corresponds to a negated existential ~∃~, represented by a line whose outermost part occurs in a negative area. Since a graph has no variables, there is no notion of substitution. Instead, the proof in Figure 16 performs the equivalent operation by connecting the two lines.

Figure 16. Proof of universal instantiation

The four steps of Figure 16 can be written in CGIF, but steps 2 and 3 require some adjustments to the coreference labels:

1. Starting graphs: **`[*y] (t ?y) ~[ [*x] ~[ (P ?x) ]]`**
2. By 2i, copy **`[*y]`** and change the defining label `*y` to a bound label `?y` in the copy:
   `[*y] (t ?y) ~[ [?y] [*x] ~[ (P ?x) ]]`
3. By 1i, insert a connection by relabeling **`*x`** and **`?x`** to **`?y`**, and erasing one copy of **`[?y]`**:
   `[*y] (t ?y) ~[ ~[ (P ?y) ]]`
4. By 3e, erase the double negation: **`[*y] (t ?y) (P ?y)`**

With the universal quantifier **`@every`**, the starting graphs of Figure 16 could be written

```
[*y] (t ?y) [(P [@every*x])]
```

The extra brackets around the last node ensure that the existential quantifier **`[*y]`** is outside the scope of `@every*x`. Universal instantiation can be used as a one-step rule to replace **`[@every*x]`** with **`?y`**. Then the brackets around **`[(P ?y)]`** may be erased to derive line 4 above.

In the *Principia Mathematica*, Whitehead and Russell proved the following theorem, which Leibniz called the *Praeclarum Theorema* (Splendid Theorem). It is one of the last and most complex theorems in propositional logic in the *Principia*, and it required a total of 43 steps:

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

With Peirce's rules, this theorem can be proved in just seven steps starting with a blank sheet of paper (Figure 17). Each step inserts or erases one graph, and the final graph is the statement of the theorem.

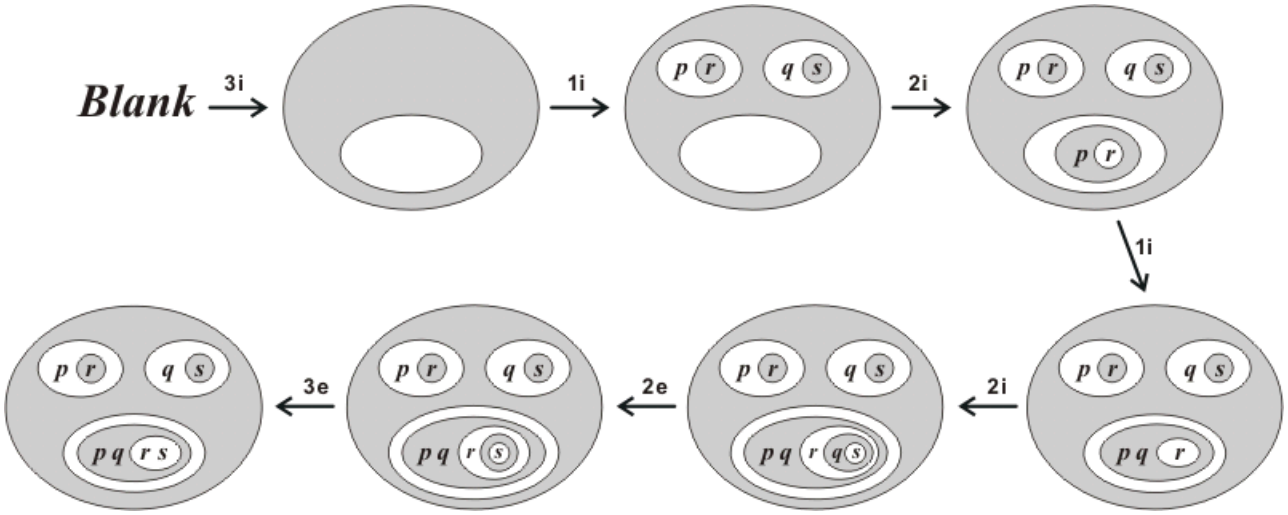

Figure 17. Proof in 7 steps instead of 43 in the *Principia*

After only four steps, the graph looks almost like the desired conclusion, except for a missing copy of *s* inside the innermost area. Since that area is positive, it is not permissible to insert *s* directly. Instead, Rule 2i copies the graph that represents $q \supset s$. By Rule 2e, the next step erases an unwanted copy of *q*. Finally, Rule 3e erases a double negation to derive the conclusion.

Unlike Gentzen's version of natural deduction, which uses a method of making and discharging assumptions, Peirce's proofs proceed in a straight line from a blank sheet to the conclusion: every step inserts or erases one subgraph in the immediately preceding graph. As Figure 17 illustrates, the first two steps of any proof that starts with a blank must draw a double negation around the blank and insert a graph into the negative area. That graph is usually the entire hypothesis of the theorem to be proved. The remainder of the proof develops the conclusion in the doubly nested blank area. Those two steps are the equivalent of Gentzen's method of making and discharging an assumption, but in Gentzen's approach, the two steps may be separated by arbitrarily many intervening steps, and a system of bookkeeping is necessary to keep track of the assumptions. With Peirce's rules, the second step follows immediately after the first, and no bookkeeping is required.

In summary, generalization and specialization, as performed by the canonical formation rules, are one-step operations that occur frequently in the ordinary use of language. Peirce's rules are also one-step operations that are simpler than the rules that Gentzen called "natural." The canonical formation rules have been implemented in nearly all CG systems, and they have been used in formal logic-based methods, informal case-based reasoning, and various computational methods. A multistep combination, called a *maximal join*, is used to determine the extent of the unifiable overlap between two CGs. In natural language processing, maximal joins help resolve ambiguities and determine the most likely connections of new information to background knowledge and the antecedents of anaphoric references. Stewart (1996) implemented Peirce's rules of inference in a first-order theorem prover for EGs and showed that its performance is comparable to resolution theorem provers. So far, no one has ever proposed a proof procedure that would have a better claim to the title "natural logic."

4. Representing Natural Language Semantics

Natural languages are highly expressive systems that can state anything that can be expressed in any formal language or logic. That enormous expressive power makes it difficult or impossible for any formalism to represent every feature of every natural language. To increase the range of expressibility, conceptual graphs constitute an open-ended family of notations with a formally defined core. Each of the following four levels of CGs can be expressed in the graphic display form or the linear CGIF:

- **Core CGs**. A typeless version of logic that expresses the full semantics of Common Logic, as defined by ISO/IEC 24707. This level corresponds to Peirce's existential graphs: its only logical primitives are conjunction, negation, and the existential quantifier. Core CGs permit quantifiers to range over relations, but Peirce also experimented with that option for EGs.
- **Extended CGs**. An upward compatible extension of the core, which adds a universal quantifier; type labels for restricting the range of quantifiers; Boolean contexts with type labels **`If`**, **`Then`**, **`Either`**, **`Or`**, **`Equivalence`**, and **`Iff`**; and the option of importing external CGIF text. Core and extended CGs have exactly the same expressive power, since the semantics of extended CGs is defined by a formal translation to core CGs. Extended CGs are usually more concise than core CGs, and they have a more direct mapping to and from natural languages.
- **CGs with contexts**. An upward compatible extension of core and extended CGs to support *metalanguage*, the option of using language to talk about language. That option requires some way of quoting or delimiting *object level* statements from *metalevel* statements. As a delimiter,

CG notations use a type of concept node, called a *context*, which contains a nested conceptual graph (Figure 18). CG contexts can be formalized by the IKL semantics, which includes the CL semantics as a subset (Hayes & Menzel 2006). Another formalization, which can be adapted to the IKL semantics, is the system of *nested graph models* (Sowa 2003, 2005). Either or both of these formalizations can be used in an upward compatible extension of core and extended CGs.

- **Research CGs**. Adaptations of the CG notation for an open-ended variety of purposes. The advantage of a standard is a fixed, reliable platform for developing computer systems and applications. The disadvantage of a standard is the inability to explore useful modifications. But the combination of an ISO standard with the option of open-ended variations gives developers a solid basis for applications without limiting the creativity of researchers. Research extensions that prove to be useful may be incorporated in some future standard.

Peirce's first use for the oval was to negate the graphs nested inside, and that is the only use supported by the ISO standard. But Peirce (1898) generalized the ovals to context enclosures, which allow relations other than negation to be linked to the enclosure. The basic use of a context enclosure is to quote the nested graphs. That syntactic option allows metalevel statements outside the context to specify how the nested (object level) graphs are interpreted. Nested graph models (NGMs) can be used to formalize the semantics of many kinds of modal and intensional logics. A hierarchy of metalevels with the NGM semantics can express the equivalent of a wide range of modal, temporal, and intentional logics. The most useful NGMs can be represented with the IKL semantics, but the many variations and their application to natural languages have not yet been fully explored.

The most common use of language about language is to talk about the beliefs, desires, and intentions of the speaker and other people. As an example, the sentence *Tom believes that Mary wants to marry a sailor*, contains three clauses, whose nesting may be marked by brackets:

```
Tom believes that [Mary wants [to marry a sailor]].
```

The outer clause asserts that Tom has a belief, which is the object of the verb *believe*. Tom's belief is that Mary wants a situation described by the nested infinitive, whose subject is the same person who wants the situation. Each clause makes a comment about the clause or clauses nested in it. References to the individuals mentioned in those clauses may cross context boundaries in various ways, as in the following two interpretations of the original English sentence:

```
Tom believes that [there is a sailor whom Mary wants [to marry]].

There is a sailor whom Tom believes that [Mary wants [to marry]].
```

The two conceptual graphs in Figure 18 represent the first and third interpretations. In the CG on the left, the existential quantifier for the concept **[Sailor]** is nested inside the situation that Mary wants. Whether such a sailor actually exists and whether Tom or Mary knows his identity are undetermined. The CG on the right explicitly states that such a sailor exists; the connections of contexts and relations imply that Tom knows him and that Tom believes that Mary also knows him. Another option (not shown) would place the concept **[Sailor]** inside the context of type **Proposition**; it would leave the sailor's existence undetermined, but it would imply that Tom believes he exists and that Tom believes Mary knows him.

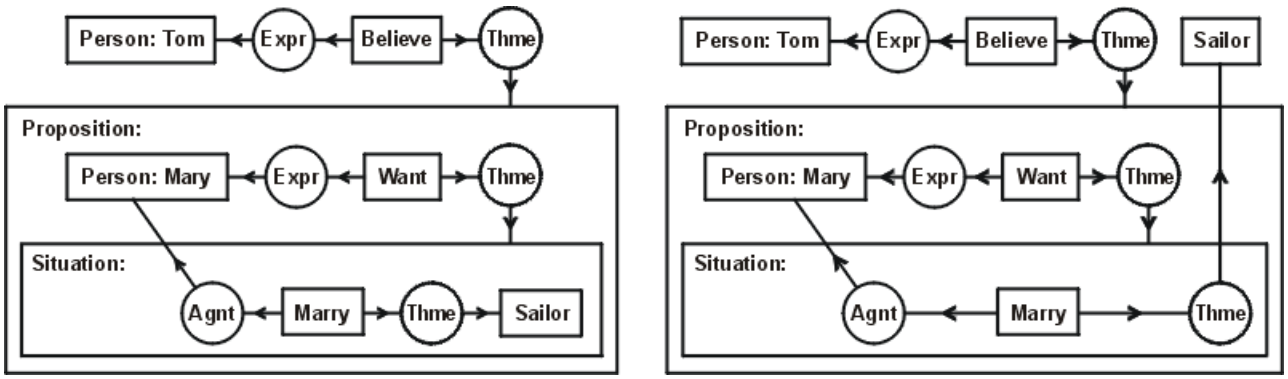

Figure 18. Two interpretations of *Tom believes that Mary wants to marry a sailor*

The context boxes illustrated in Figures 4 and 6 express negations or operators such as **`If`** and **`Then`**, which are defined in terms of negations. But the contexts of type **`Proposition`** and **`Situation`** in Figure 18 raise new issues of logic and ontology. The CL semantics can represent entities of any type, including propositions and situations, but it has no provision for relating such entities to the internal structure of CL sentences. A more expressive language, called IKL (Hayes & Menzel 2006), was defined as an upward compatible extension of CL. The IKL semantics introduces entities called *propositions* and a special operator, spelled **`that`**, which relates IKL sentences to the propositions they express. IKL semantics does not have a built-in type for situations, but it is possible in IKL to make statements that state the existence of entities of type **`Situation`** and relate them to propositions.

The first step toward translating the CGs in Figure 18 to IKL is to write them in an extended version of CGIF, which allows CGs to be nested inside concept nodes of type **`Proposition`** or **`Situation`**. Following is the CGIF for the CG on the left:

```
[Person: Tom] [Believe: *x1] (Expr ?x1 Tom)
(Thme ?x1 [Proposition:
   [Person: Mary] [Want: *x2] (Expr ?x2 Mary)
   (Thme ?x2 [Situation:
      [Marry: *x3] [Sailor: *x4] (Agnt ?x3 Mary) (Thme ?x3 ?x4)])])
```

This statement uses the option of moving the concept nodes for the types **`Proposition`** and **`Situation`** inside the relation nodes of type **`Thme`**. That option has no semantic significance, but it makes the order of writing the CGIF closer to English word order. A much more important semantic question is the relation between situations and propositions. In the ontology commonly used with CGs, that relation is spelled **`Dscr`** and called the *description relation*. The last two lines of the CGIF statement above could be rewritten in the following form:

```
(Thme ?x2 [Situation: *s]) (Dscr ?s [Proposition:
   [Marry: *x3] [Sailor: *x4] (Agnt ?x3 Mary) (Thme ?x3 ?x4)])])
```

The last line is unchanged, but the line before it states that the theme of `x2` is the situation `s` and the description of `s` is the proposition stated on the last line. In effect, every concept of type `Situation` that contains a nested CG is an abbreviation for a situation that is described by a concept of type `Proposition` that has the same nested CG. This expanded CGIF statement can then be translated to IKL (which is based on CLIF syntax with the addition of the operator **`that`**).

```
(exists ((x1 Believe)) (and (Person Tom) (Expr x1 Tom)
   (Thme x1 (that (exists ((x2 Want) (s Situation))
      (and (Person Mary) (Expr x2 Mary) (Thme x2 s) (Dscr s (that
         (exists ((x3 Marry) (x4 Sailor)) (and (Agnt x3 Mary) (Thme x3 x4)
))))))))))
```

Note that every occurrence of **Proposition** in CGIF corresponds to **that** in IKL. The syntax of CLIF or IKL requires more parentheses than CGIF because every occurrence of **exists** or **and** requires an extra closing parenthesis at the end.

As these examples illustrate, the operator **that** adds an enormous amount of expressive power, but IKL still has a first-order style of semantics. The proposition nodes in CGs or the **that** operator in IKL introduce abstract entities of type **Proposition**, but propositions are treated as zero-argument relations, which are supported by the semantics of Common Logic. Although language about propositions is a kind of metalanguage, it does not, by itself, go beyond first-order logic. Tarski (1933), for example, demonstrated how a stratified series of metalevels, each of which is purely first order, can be used without creating paradoxes or going beyond the semantics of FOL. In effect, Tarski avoided paradoxes by declaring that certain kinds of sentences (those that violate the stratification) do not express propositions in his models. The IKL model theory has a similar way of avoiding paradoxes: it does not require every model to include a proposition for every possible sentence. For example, the following English sentence, which sounds paradoxical, could be expressed in either IKL or CGIF syntax:

> *There exists a proposition p, p is true,*
> *and p is the proposition that p is false.*

Since IKL does not require every sentence to express a proposition in every model, there are permissible IKL models in which this sentence is false simply because no such proposition exists. Therefore, the paradox vanishes because the sentence has a stable, but false, truth value.

Issues of context and metalanguage require some syntactic and semantic extensions beyond the CL standard. The syntax for context was proposed by Sowa (1984) with a semantics that was a subset of the later IKL and NGM versions. Syntax for the following three extensions has also been available since 1984, and some CG systems have implemented versions of them. But they are not yet in ISO standard CGIF because a fully general treatment would involve ongoing research in linguistics:

- **Generalized quantifiers.** In addition to the usual quantifiers of *every* and *some*, natural languages support an open-ended number of quantificational expressions, such as *exactly one*, *at least seven*, or *considerably more*. Some of these quantifiers, such as *exactly one cat*, could be represented as **[Cat: @1]** and defined in terms of the CL standard. Others, such as *at least seven cats*, could be represented **[Cat: @≤7]** and defined with a version of set theory added to the base logic. But quantifiers such as *considerably more* would require some method of approximate reasoning, such as fuzzy sets or rough sets.
- **Indexicals.** Peirce observed that every statement in logic requires at least one indexical to fix the referents of its symbols. The basic indexical, which corresponds to the definite article *the*, is represented by the symbol # inside a concept node: **[Dog: #]** would represent the phrase *the dog*. The pronouns *I*, *you*, and *she* would be represented **[Person: #I]**, **[Person: #you]**, and **[Person: #she]**. To process indexicals, some linguists propose versions of *dynamic semantics*, in which the model is updated during the discourse. A simpler method is to treat the # symbol as a syntactic marker that indicates a incomplete interpretation of the original sentence. With this approach, the truth value of a CG that contains any occurrences of # is not determined until those markers are replaced by names or coreference labels. This approach supports indexicals in an intermediate representation, but uses a conventional model theory to evaluate the final resolution.
- **Plural nouns.** Plurals have been represented in CGs by set expressions inside the concept boxes. The concept **[Cat: {*}@3]** would represent *three cats*, and **[Dog: {Lucky,**

Macula}] would represent *the dogs Lucky and Macula*. Various methods have been proposed for representing distributed and collective plurals and translating them to versions of set theory and mereology.

Simple versions of these features have been implemented in CG systems. The difficult issues involve generalizing them in a systematic way to cover all the variations that occur in natural languages.

5. Ongoing Research

The major difference between natural languages and formal languages is not in the notation. For the world's first formal logic, Aristotle used a subset of Greek. Some computer systems use a *controlled natural language* based on a formally defined subset of a natural language. The defining characteristic of a formal language is that the meaning of any statement is completely determined by its form or syntax. Natural languages, however, are highly context dependent. Ambiguities and indexicals are two kinds of dependencies that have been thoroughly analyzed, but vagueness is more challenging. Peirce (1902:748) defined vagueness in an article for Baldwin's *Dictionary of Philosophy and Psychology*:

> A proposition is vague when there are possible states of things concerning which it is intrinsically uncertain whether, had they been contemplated by the speaker, he would have regarded them as excluded or allowed by the proposition. By intrinsically uncertain we mean not uncertain in consequence of any ignorance of the interpreter, but because the speaker's habits of language were indeterminate.

In other writings, Peirce explained that a vague statement requires some additional information to make it precise, but the sentence itself gives little or no indication of what kind of information is missing. That characteristic distinguishes vagueness from the ambiguity illustrated in Figure 18. An ambiguous sentence has two or more interpretations, and the kind of information needed to resolve the ambiguity can usually be determined by analyzing the sentence itself. An indexical is a word or phrase, such as *she* or *the book*, whose referent is not specified, but the kind of thing that would serve as a referent and the means of finding it are usually determined. But the meaning of a vague sentence, as Peirce observed, is "intrinsically uncertain" because the method for finding the missing information is unknown, perhaps even to the speaker.

Peirce's definition is compatible with an approach to vagueness by the logical methods of *underspecification* and *supervaluation*. As an example, the graph on the left of Figure 18 is underspecified because it is true in all three of the interpretations of the sentence *Tom believes that Mary wants to marry a sailor.* The graph on the right is the most specific because it is true in only one interpretation. CGs represent indexicals with underspecified concepts such as **[Person: #she]** for *she* and **[Book: #]** for *the book*, whose referents could later be resolved by the methods of discourse representation theory. To accommodate the *truth-value gaps* of vague sentences that are neither true nor false, van Fraassen (1966) proposed supervaluation as a model-theoretic method that allows some sentence *p* to have an indeterminate truth value in a specific model. But other sentences, such as $\boldsymbol{p \vee \sim p}$ would have a supervalue of true in all models, while the sentence $\boldsymbol{p \wedge \sim p}$ would have the supervalue false in all models.

For language understanding, Hintikka (1973:129) observed that infinite families of models might be theoretically interesting, but "It is doubtful whether we can realistically expect such structures to be somehow actually involved in our understanding of a sentence or in our contemplation of its meaning." As an alternative, he proposed a *surface model* of a sentence S as "a mental anticipation of what can happen in one's step-by-step investigation of a world in which S is true." Instead of a logic of

vagueness, Hintikka suggested a method of constructing a model that would use all available information to fill in the details left indeterminate in a given text. The first stage of constructing a surface model begins with the entities occurring in a sentence or story. During the construction, new facts may be asserted that block certain extensions or facilitate others. A conventional model is the limit of a surface model that has been extended infinitely far, but such infinite processes are not required for normal understanding.

After forty years, supervaluations are still widely used by logicians, but Fodor and Lepore (1996) denounced them as useless for linguistics and psychology. Surface models, although largely neglected, can be constructed by constraint satisfaction methods similar to the heuristic techniques in AI and *dynamic semantics* in linguistics. Conceptual graphs are convenient for such methods because they can be used to represent arbitrary statements in logic, to represent formal models of those statements, and to represent each step from an initially vague statement to a complete specification. Any Tarski-style model, for example, can be represented as a potentially infinite simple graph, whose only logical operators are conjunction and existential quantification. Each step in constructing a surface model is a subgraph of a potentially infinite complete model, but for most applications, there is no need to finish the construction. Such techniques have been successfully used for understanding dialogs, in which fragments of information from multiple participants must be assembled to construct a larger view of a subject.

Implementations, in either computers or neurons, can exploit the topological properties of graphs, which include symmetry, duality, connectivity, and cycles. For knowledge representation, the topology often reveals similarities that are masked by different choices of ontology. Figure 19 shows two different ways of representing the sentence *Sue gives a child a book*. The CG on the left represents the verb by a triadic relation, while the one on the right represents it by a concept linked to three dyadic relations: agent, theme, and recipient. Different ontologies lead to different numbers of concept and relation nodes with different type labels, but a characteristic invariant of giving is the triadic connectivity of three participants to a central node.

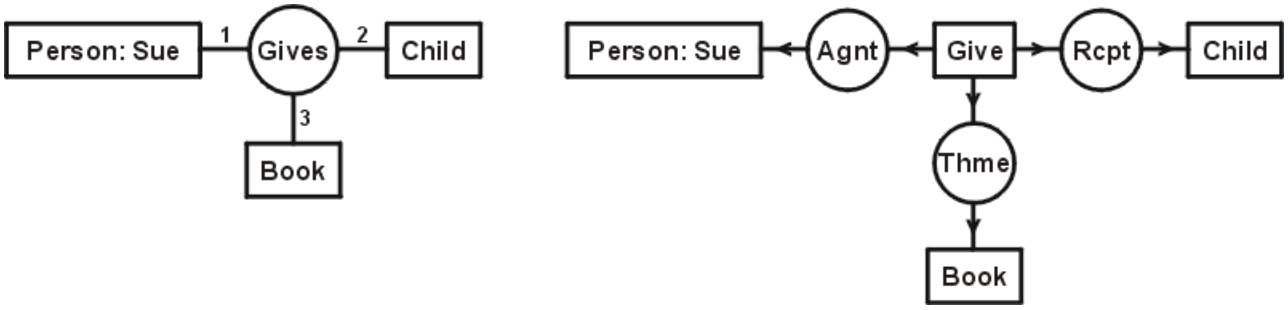

Figure 19. Two ways of representing an act of giving

Although the relation type **`Gives`** and the concept type **`Give`** have similar names, concept nodes and relation nodes cannot be directly mapped to one another. But when the topology is analyzed, adjacent nodes can be merged to reduce the size of the graphs while preserving invariants such as cycles and connectivity. The invariants can indicate common underlying relationships expressed with different ontologies. Sowa and Majumdar (2003) presented a more complex example that compared one CG derived from a relational database to another CG derived from an English sentence, both of which represented the same physical structure. The database CG had 15 concept nodes and 9 relation nodes, but the CG from English had 12 concept nodes and 11 relation nodes. Furthermore, no type label on any node of one CG was identical to any type label on any node of the other. Given only the constraint that five concept nodes of each CG referred to the same five objects, the algorithms correctly derived the mappings from one ontology to the other. Those mappings could then be used to compare other

representations with the same two ontologies. Ongoing research on methods for representing graphs has led to high-performance algorithms for searching, comparing, and transforming large numbers of very large graphs.

As an example of applied research, one of the largest commercial CG systems is Sonetto (Sarraf & Ellis 2006), which uses extended versions of earlier algorithms by Levinson and Ellis (1992). A key innovation of Sonetto is its semi-automated methods for extracting ontologies and business rules from unstructured documents. The users who assist Sonetto in the knowledge extraction process are familiar with the subject matter, but they have no training in programming or knowledge engineering. CGIF is the knowledge representation language for ontologies, rules, and queries. It is also used to manage the schemas of documents and other objects in the system and to represent the rules that translate CGIF to XML and other formats. For the early CG research, see the collections edited by Nagle et al. (1992), Way (1992), and Chein (1996). More recent research on CGs has been published in the annual proceedings of the International Conferences on Conceptual Structures.

Appendix: The Common Logic Standard

Common Logic (CL) evolved from two projects to develop parallel ANSI standards for conceptual graphs and the Knowledge Interchange Format (Genesereth & Fikes 1992). Eventually, those projects were merged into a single ISO project to develop a common abstract syntax and model-theoretic foundation for a family of logic-based notations (ISO/IEC 24707). Hayes and Menzel (2001) defined a very general model theory for CL, which Hayes and McBride (2003) used to define the semantics for the languages RDF(S) and OWL. In addition to the abstract syntax and model theory, the CL standard specifies three concrete dialects that are capable of expressing the full CL semantics: the Common Logic Interchange Format (CLIF), the Conceptual Graph Interchange Format (CGIF), and the XML-based notation for CL (XCL). Since the semantics of RDF and OWL is based on a subset of CL semantics, those languages can also be considered dialects of CL: any statement in RDF or OWL can be translated to CLIF, CGIF, or XCL, but only a subset of CL can be translated back to RDF or OWL.

The CL syntax allows quantifiers to range over functions and relations, but CL retains a first-order style of model theory and proof theory. To support a higher-order syntax, but without the computational complexity of higher-order semantics, the CL model theory uses a single domain D that includes individuals, functions, and relations. The option of limiting the domain of quantification to a single set was suggested by Quine (1954) and used in various theorem provers that allow quantifiers to range over relations (Chen et al., 1993).

The CL standard is defined in an abstract syntax that is independent of any concrete notation. It does, however, support the full Unicode character set and the URIs of the Semantic Web. The three dialects defined in the standard (CLIF, CGIF, and XCL) use only the ASCII subset of Unicode for their basic syntax, but they allow any Unicode symbols in names and character strings. Although CGIF and CLIF had different origins, the two notations have many similarities. As an example, following is the core CGIF for the sentence *Bob drives his Chevy to St. Louis*:

```
[*x] [*y]
(Drive ?x) (Person Bob) (City "St. Louis") (Chevy ?y)
(Agnt ?x Bob) (Dest ?x "St. Louis") (Thme ?x ?y) (Poss Bob ?y)
```

In core CGIF, the concept nodes **[*x]** and **[*y]** represent the existential quantifiers ∃*x* and ∃*y*. Following is the CLIF statement, which uses the keyword `exists` to introduce a list of existentially quantified variables:

```
(exists (x y)
   (and (Drive x) (Person Bob) (City "St. Louis") (Chevy y)
        (Agnt x Bob) (Dest x "St. Louis") (Thme x y) (Poss Bob y) ))
```

Although CGIF and CLIF look similar, there are several fundamental differences:

1. Since CGIF is a serialized representation of a graph, labels such as `x` or `y` represent coreference links between nodes, but they represent variables in CLIF or predicate calculus.
2. CGIF distinguishes the labels `*x` and `?x` from a name like `Bob` by an explicit prefix. CLIF, however, has no special markers on variables; the only distinction is that variables appear in a list after the quantifier.
3. Since the nodes of a graph have no inherent ordering, a CGIF sentence is an unordered list of nodes. Unless grouped by context brackets, the list may be permuted without affecting the semantics.
4. The CLIF operator `and` does not occur in CGIF because the conjunction of nodes within any context is implicit. Omitting the conjunction operator in CGIF tends to reduce the number of parentheses.

Extended CGIF allows monadic relation names to be used as type labels, and extended CLIF allows them to be used as restrictions on the scope of quantifiers. Following is the extended CGIF for the above sentence:

```
[Drive *x] [Person: Bob] [City: "St. Louis"] [Chevy *y]
(Agnt ?x Bob) (Dest ?x "St. Louis") (Thme ?x ?y) (Poss Bob ?y)
```

And following is the equivalent in extended CLIF:

```
(exists ((x Drive) (y Chevy))
   (and (Person Bob) (City "St. Louis") (Agnt x Bob)
   (Dest x "St. Louis") (Thme x y) (Poss Bob y) ))
```

Since the semantics of any statement in extended CGIF and CLIF is defined by its translation to the core language, neither language makes a semantic distinction between type labels and monadic relations. If a statement in a strongly typed language, such as the Z Specification Language, is translated to CGIF or CLIF, the Z types are mapped to CGIF type labels or CLIF quantifier restrictions. A syntactically correct statement in Z and its translation to CGIF or CLIF have the same truth value. But an expression with a type mismatch would cause a syntax error in Z, but it would merely be false in CGIF or CLIF.

As another example, Figure 20 shows a CG for the sentence *If a cat is on a mat, then it is a happy pet.* The dotted line that connects the concept **[Cat]** to the concept **[Pet]**, which is called a *coreference link*, indicates that they both refer to the same entity. The **Attr** relation indicates that the cat, also called a pet, has an attribute, which is an instance of happiness.

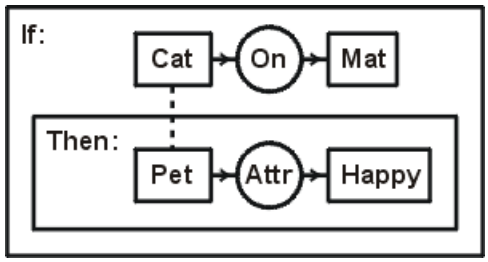

Figure 20. CG display form for *If a cat is on a mat, then it is a happy pet.*

The dotted line in Figure 20, called a *coreference link*, is shown in CGIF by the defining label ***x** in the concept **[Cat: *x]** and the bound label **?x** in **[Pet: ?x]**. Following is the extended CGIF:

```
[If: [Cat *x] [Mat *y] (On ?x ?y)
   [Then: [Pet ?x] [Happy *z] (Attr ?x ?z) ]]
```

In CGs, functions are represented by conceptual relations called *actors*. Figure 21 is the CG display form for the following equation written in ordinary algebraic notation:

```
y = (x + 7)/sqrt(7)
```

The three functions in this equation would be represented by three actors, which are shown in Figure 21 as diamond-shaped nodes with the type labels **Add**, **Sqrt**, and **Divide**. The concept nodes contain the input and output values of the actors. The two empty concept nodes contain the output values of **Add** and **Sqrt**.

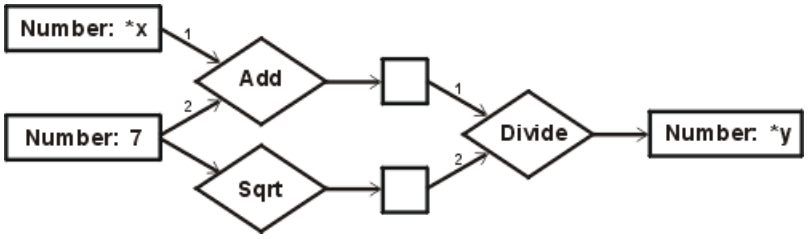

Figure 21. CL functions represented by actor nodes

In CGIF, actors are represented as relations with two kinds of arcs: a sequence of *input arcs* separated by a vertical bar from a sequence of *output arcs*.

```
[Number: *x] [Number: *y] [Number: 7]
(Add ?x 7 | [*u]) (Sqrt 7 | [*v]) (Divide ?u ?v | ?y)
```

In the display form, the input arcs of `Add` and `Divide` are numbered 1 and 2 to indicate the order in which the arcs are written in CGIF. Following is the corresponding CLIF:

```
(exists ((x Number) (y Number))
   (and (Number 7) (= y (Divide (Add x 7) (Sqrt 7)))))
```

No CLIF variables are needed to represent the coreference labels `*u` and `*v` since the functional notation used in CLIF shows the connections directly.

CLIF only permits functions to have a single output, but extended CGIF allows actors to have multiple outputs. The following actor of type **IntegerDivide** has two inputs: an integer **x** and an integer **7**. It also has two outputs: a quotient **u** and a remainder **v**.

```
(IntegerDivide [Integer: *x] [Integer: 7] | [*u] [*v])
```

When this actor is translated to core CGIF or CLIF, the vertical bar is removed, and the actor becomes an ordinary relation with four arguments; the distinction between inputs and outputs is lost. In order to assert the constraint that the last two arguments are functionally dependent on the first two arguments, the following CGIF sentence asserts that there exist two functions, identified by the coreference labels `Quotient` and `Remainder`, which for every combination of input and output values are logically equivalent to an actor of type **IntegerDivide** with the same input and output values:

```
[Function: *Quotient] [Function: *Remainder]
[[@every*x1] [@every*x2] [@every*x3] [@every*x4]
[Equiv: [Iff: (IntegerDivide ?x1 ?x2 | ?x3 ?x4)]
        [Iff: (#?Quotient ?x1 ?x2 | ?x3) (#?Remainder ?x1 ?x2 | ?x4)]]]
```

Each line of this example illustrates one or more features of CGIF. The first line represents existential quantifiers for two entities of type **Function**. On the second line, the context bracket [encloses the concept nodes with universal quantifiers, marked by **@every**, to show that the existential quantifiers for **Quotient** and **Remainder** include the universals within their scope. The equivalence on lines three and four shows that an actor of type **IntegerDivide** is logically equivalent to a conjunction of the quotient and remainder functions. Finally, the symbol # on line four shows that the coreference labels **?Quotient** and **?Remainder** are being used as type labels. Following is the corresponding CLIF:

```
(exists ((Quotient Function) (Remainder Function))
   (forall (x1 x2 x3 x4)
      (iff (IntegerDivide x1 x2 x3 x4)
           (and (= x3 (Quotient x1 x2)) (= x4 (Remainder x1 x2))))))
```

As another example of quantifiers that range over relations, someone might say "Bob and Sue are related," but not say exactly how they are related. The following sentences in CGIF and CLIF state that there exists some familial relation *r* that relates Bob and Sue:

```
[Relation: *r] (Familial ?r) (#?r Bob Sue)

(exists ((r Relation)) (and (Familial r) (r Bob Sue)))
```

The concept **[Relation: *r]** states that there exists a relation *r*. The next two relations state that *r* is familial and *r* relates Bob and Sue.

This brief survey has illustrated nearly every major feature of CGIF and CLIF. One important feature that has not been mentioned is a *sequence marker* to support relations with a variable number of arguments. Another is the use of comments, which can be placed before, after, or inside any concept or relation node in CGIF. The specifications in the CL standard guarantee that any sentence expressed in the dialects CGIF, CLIF, or XCL can be translated to any of the others in a logically equivalent form. Although the translation will preserve the semantics, it is not guaranteed to preserve all syntactic details: a sentence translated from one dialect to another and then back to the first will be logically equivalent to the original, but some subexpressions might be reordered or replaced by semantic equivalents. Following is a summary of the CGIF grammar; see ISO/IEC 24707 for the complete specification of the Common Logic syntax and semantics.

Lexical Grammar Rules

The syntax rules are written in Extended Backus-Naur Form (EBNF) rules, as specified by ISO/IEC 14977. The CGIF syntax rules assume the same four types of names as CLIF: **namecharsequence** for names not enclosed in quotes; **enclosedname** for names enclosed in double quotes; **numeral** for numerals consisting of one or more digits; and **quotedstring** for character strings enclosed in single quotes. But because of syntactic differences between CGIF and CLIF, CGIF must enclose more names in quotes than CLIF in order to avoid ambiguity. Therefore, the only CG names not enclosed in quotes belong to the categories **identifier** and **numeral**.

```
CGname = identifier | '"', (namecharsequence - identifier), '"'
         | numeral | enclosedname | quotedstring;

identifier = letter, {letter | digit | "_"};
```

When CGIF is translated to CL, a CG name is translated to a CLIF name by removing any quotes around a name character sequence. CLIF does not make a syntactic distinction between constants and variables, but in CGIF any CG name that is not used as a defining label or a bound label is called a

constant. The start symbol of CGIF syntax is the category **`text`** for a complete text or the category `CG` for just a single conceptual graph.

Core CGIF Grammar Rules

An *actor* is a conceptual relation that represents a function in Common Logic. It begins with **`(`**, an optional comment, an optional string **`#?`**, a CG name, **`|`**, an arc, an optional end comment, and **`)`**. If the CG name is preceded by **`#?`**, it represents a bound coreference label; otherwise, it represents a type label. The arc sequence represents the arguments of the CL function and the last arc represents the value of the function.

```
actor = "(", [comment], ["#?"], CGname, arcSequence, "|", arc,
            [endComment], ")";
```

An *arc* is an optional comment followed by a reference. It links an actor or a conceptual relation to a concept that represents one argument of a CL function or relation.

```
arc = [comment], reference;
```

An *arc sequence* is a sequence of zero or more arcs, followed by an option consisting of an optional comment, **`?`**, and a sequence marker.

```
arcSequence = {arc}, [[comment], "?", seqmark];
```

A *comment* or an *end comment* is a character string that has no effect on the semantics of a conceptual graph or any part of a conceptual graph. A comment begins with **`"/*"`**, followed by a character string that contains no occurrence of **`"*/"`**, and ends with **`"*/"`**. A comment may occur immediately after the opening bracket of any concept, immediately after the opening parenthesis of any actor or conceptual relation, immediately before any arc, or intermixed with the concepts and conceptual relations of a conceptual graph. An end comment begins with **`;`**, followed by a character string that contains no occurrence of **`]`** or **`)`**. An end comment may occur immediately before the closing bracket of any concept or immediately before the closing parenthesis of any actor or conceptual relation.

```
comment = "/*", {(character-"*") | ["*", (character-"/")]}, ["*"], "*/";

endComment = ";", {character - ("]" | ")")};
```

A *concept* is either a context, an existential concept, or a coreference concept. Every concept begins with `[` and an optional comment; and every concept ends with an optional end comment and `]`. Between the beginning and end, a context contains a CG; an existential concept contains `*` and either a CG name or a sequence marker; and a coreference concept contains **`:`** and a sequence of one or more references. A context that contains a blank CG is said to be *empty*, even if it contains one or more comments; any comment that occurs immediately after the opening bracket shall be part of the concept, not the following CG.

```
concept = "[", [comment],
           (CG | "*", (CGname | seqmark) | ":", {reference}- ),
           [endComment], "]";
```

A *conceptual graph* (CG) is an unordered list of concepts, conceptual relations, negations, and comments.

```
CG = {concept | conceptualRelation | negation | comment};
```

A *conceptual relation* is either an ordinary relation or an actor. An ordinary relation, which represents a CL relation, begins with **`(`**, an optional comment, an optional string **`#?`**, a CG name, an optional end

comment, and **)**. If the CG name is preceded by **#?**, it represents a bound coreference label; otherwise, it represents a type label. An ordinary relation has just one sequence of arcs, but an actor has two sequences of arcs.

```
conceptualRelation = ordinaryRelation | actor;

ordinaryRelation = "(", [comment], ["#?"], CGname, arcSequence,
                        [endComment], ")";
```

A *negation* is **~** followed by a context.

```
negation = "~", context;
```

A *reference* is an optional **?** followed by a CG name. A CG name prefixed with **?** is called a *bound coreference label*; without the prefix **?**, it is called a *constant*.

```
reference = ["?"], CGname;
```

A *text* is a context, called an *outermost context*, that has an optional name, has an arbitrarily large conceptual graph, and is not nested inside any other context. It consists of **[**, an optional comment, the type label **Proposition**, **:**, an optional CG name, a conceptual graph, an optional end comment, and **]**. Although a text may contain core CGIF, the type label Proposition is outside the syntax of core CGIF.

```
text = "[", [comment], "Proposition", ":", [CGname], CG,
            [endComment], "]";
```

Extended CGIF Grammar Rules

Extended CGIF is superset of core CGIF, and every syntactically correct sentence of core CGIF is also syntactically correct in extended CGIF. Its most prominent feature is the option of a *type label* or a *type expression* on the left side of any concept. In addition to types, extended CGIF adds the following features to core CGIF:

- More options in concepts, including universal quantifiers.
- Boolean contexts for representing the operators or, if, and iff.
- The option of allowing concept nodes to be placed in the arc sequence of conceptual relations.
- The ability to import text into a text.

These extensions are designed to make sentences more concise, more readable, and more suitable as a target language for translations from natural languages and from other CL dialects, including CLIF. None of them, however, extend the expressive power of CGIF beyond the CG core, since the semantics of every extended feature is defined by its translation to core CGIF, whose semantics is defined by its translation to the abstract syntax of Common Logic.

The following grammar rules of extended CGIF have the same definitions as the core CGIF rules of the same name: **arcSequence**, **conceptualRelation**, **negation**, **ordinaryRelation**, **text**. The following grammar rules of extended CGIF don't occur in core CGIF, or they have more options than the corresponding rules of core CGIF: **actor**, **arc**, **boolean**, **CG**, **concept**, **eitherOr**, **equivalence**, **ifThen**, **typeExpressio**n.

An *actor* in extended CGIF has the option of zero or more arcs following **|** instead of just one arc.

```
actor = "(", [comment], ["#?"], CGname,
        arcSequence, "|", {arc}, [endComment], ")";
```

An *arc* in extended CGIF has the options of a defining coreference label and a concept in addition to a bound coreference label.

```
arc = [comment], (reference | "*", CGname | concept);
```

A *boolean* is either a negation or a combination of negations that represent an either-or construction, an if-then construction, or an equivalence. Instead of being marked with **~**, the additional negations are represented as contexts with the type labels **`Either`**, **`Or`**, **`If`**, **`Then`**, **`Equiv`**, **`Equivalence`**, or **`Iff`**.

```
boolean = negation | eitherOr | ifThen | equivalence;
```

A *concept* in extended CGIF permits any combination allowed in core CGIF in the same node and it adds two important options: a type field on the left side of the concept node, and a universal quantifier on the right. Four options are permitted in the type field: a type expression, a bound coreference label prefixed with **`"#"`**, a constant, or the empty string; a colon is required after a type expression, but optional after the other three.

```
concept = "[", [comment],
          ( (typeExpression, ":"
            | ["#?"], CGname, [":"]),
            [["@every"], "*", CGname], {reference}, CG
          | ["@every"], "*", seqmark
          ), [endComment], "]";
```

A *conceptual graph* (CG) in extended CGIF adds Boolean combinations of contexts to core CGIF.

```
CG = {concept | conceptualRelation | boolean | comment};
```

An *either-or* is a negation with a type label **`Either`** that contains zero or more negations with a type label **`Or`**.

```
eitherOr = "[", [comment], "Either", [":"],
                {"[", [comment], "Or", [":"], CG, [endComment], "]"}
                [endComment], "]";
```

An *equivalence* is a context with a type label **`Equivalence`** or **`Equiv`** that contains two contexts with a type label **`Iff`**. It is defined as a pair of if-then constructions, each with one of the iff-contexts as antecedent and the other as consequent.

```
equivalence = "[", [comment], ("Equivalence" | "Equiv"), [":"],
                  "[", [comment], "Iff", [":"], CG, [endComment], "]",
                  "[", [comment], "Iff", [":"], CG, [endComment], "]",
                   [endComment], "]";
```

An *if-then* is a negation with a type label **`If`** that contains a negation with a type label **`Then`**.

```
ifThen = "[", [comment], "If", [":"], CG,
                "[", [comment], "Then", [":"], CG, [endComment], "]",
                [endComment], "]";
```

A *type expression* is a lambda-expression that may be used in the type field of a concept. The symbol **`@`** marks a type expression, since the Greek letter **λ** is not available in the ASCII subset of Unicode.

```
typeExpression = "@", "*", CGname, CG;
```

References

Chein, Michel, ed., (1996) *Revue d'intelligence artificielle*, Numéro Spécial Graphes Conceptuels, vol. 10, no. 1.

Dau, Frithjof (2006) "Some notes on proofs with Alpha graphs," in Schärfe et al., pp. 172-188.

Davidson, Donald (1967) "The logical form of action sentences," reprinted in D. Davidson (1980) *Essays on Actions and Events*, Clarendon Press, Oxford, pp. 105-148.

Fillmore, Charles J. (1968) "The case for case" in E. Bach & R. T. Harms, eds., *Universals in Linguistic Theory*, Holt, Rinehart and Winston, New York, 1-88.

Fodor, Jerry A., & Ernie Lepore (1996) "What can't be valued, can't be valued, and it can't be supervalued either," *Journal of Philosophy* **93**, 516-536.

Genesereth, Michael R., & Richard Fikes, eds. (1992) *Knowledge Interchange Format, Version 3.0 Reference Manual*, TR Logic-92-1, Computer Science Department, Stanford University.

Gentzen, Gerhard (1935) "Untersuchungen über das logische Schließen," translated as "Investigations into logical deduction" in *The Collected Papers of Gerhard Gentzen*, ed. and translated by M. E. Szabo, North-Holland Publishing Co., Amsterdam, 1969, pp. 68-131.

Hayes, Patrick (2005) "Translating semantic web languages into Common Logic," http://www.ihmc.us/users/phayes/CL/SW2SCL.html

Hayes, Patrick, & Brian McBride (2003) "RDF semantics," W3C Technical Report, http://www.w3.org/TR/rdf-mt/

Hayes, Patrick, & Chris Menzel (2001) "A semantics for the Knowledge Interchange Format," *Proc. IJCAI 2001 Workshop on the IEEE Standard Upper Ontology*, Seattle.

Hayes, Patrick, & Chris Menzel (2006) "IKL Specification Document," http://www.ihmc.us/users/phayes/IKL/SPEC/SPEC.html

Hays, David G. (1964) "Dependency theory: a formalism and some observations," *Language* **40:4**, 511-525.

Hintikka, Jaakko (1973) "Surface semantics: definition and its motivation," in H. Leblanc, ed., *Truth, Syntax and Modality*, North-Holland, Amsterdam, 128-147.

Hughes, Dominic J. D. (2006) "Proofs Without Syntax," *Annals of Mathematics*,

ISO/IEC (2002) *Z Formal Specification Notation — Syntax, Type System, and Semantics*, IS 13568, International Organisation for Standardisation.

ISO/IEC (2006) *Common Logic (CL) — A Framework for a family of Logic-Based Languages*, FCD 24707, International Organisation for Standardisation.

Kamp, Hans (1981) "A theory of truth and semantic representation," in *Formal Methods in the Study of Language*, ed. by J. A. G. Groenendijk, T. M. V. Janssen, & M. B. J. Stokhof, Mathematical Centre Tracts, Amsterdam, 277-322.

Kamp, Hans, & Uwe Reyle (1993) *From Discourse to Logic*, Kluwer, Dordrecht.

Klein, Sheldon, & Robert F. Simmons (1963) "Syntactic dependence and the computer generation of coherent discourse," *Mechanical Translation* **7**.

Levinson, Robert A., & Gerard Ellis (1992) "Multilevel hierarchical retrieval," *Knowledge Based Systems* **5:3**, pp. 233-244.

McKinley, Richard (2006) *Categorical Models of First-Order Classical Proofs*, PhD Dissertation, University of Bath.

Nagle, T. E., J. A. Nagle, L. L. Gerholz, & P. W. Eklund, eds. (1992) *Conceptual Structures: Current Research and Practice*, Ellis Horwood, New York.

Newell, Allen, & Herbert A. Simon (1972) *Human Problem Solving*, Prentice-Hall, Englewood Cliffs, NJ.

Peano, Giuseppe (1889) *Aritmetices principia nova methoda exposita*, Bocca, Torino.

Peirce, Charles Sanders (1870) "Description of a notation for the logic of relatives," reprinted in *Writings of Charles S. Peirce*, Indiana University Press, Bloomington, vol. 2, pp. 359-429.

Peirce, Charles Sanders (1880) "On the algebra of logic," *American Journal of Mathematics* **3**, 15-57.

Peirce, Charles Sanders (1885) "On the algebra of logic," *American Journal of Mathematics* **7**, 180-202.

Peirce, Charles Sanders (1898) *Reasoning and the Logic of Things*, The Cambridge Conferences Lectures of 1898, ed. by K. L. Ketner, Harvard University Press, Cambridge, MA, 1992.

Peirce, Charles Sanders (1902) "Vague," in J. M. Baldwin, ed., *Dictionary of Philosophy and Psychology*, MacMillan, New York, p. 748.

Peirce, Charles Sanders (1906) Manuscripts on existential graphs, *Collected Papers of Charles Sanders Peirce*, vol. 4, Harvard University Press, Cambridge, MA, pp. 320-410.

Peirce, Charles Sanders (1909) Manuscript 514, with commentary by J. F. Sowa, http://www.jfsowa.com/peirce/ms514.htm

Quillian, M. Ross (1966) "Semantic memory," in M. Minsky, ed., *Semantic Information Processing*, MIT Press, Cambridge, MA, pp. 227-270.

Quine, Willard Van Orman (1954) "Reduction to a dyadic predicate," *J. Symbolic Logic* **19**, reprinted in W. V. Quine, *Selected Logic Papers*, Enlarged Edition, Harvard University Press, Cambridge, MA, 1995, pp. 224-226.

Sarraf, Qusai, & Gerard Ellis (2006) "Business Rules in Retail: The Tesco.com Story," *Business Rules Journal* **7:6**, http://www.brcommunity.com/a2006/n014.html

Schärfe, Henrik, Pascal Hitzler, & Peter Øhrstrom, eds. (2006) *Conceptual Structures: Inspiration and Application*, LNAI 4068, Springer, Berlin.

Schank, Roger C., ed. (1975) *Conceptual Information Processing*, North-Holland Publishing Co., Amsterdam.

Selz, Otto (1913) *Über die Gesetze des geordneten Denkverlaufs*, Spemann, Stuttgart.

Selz, Otto (1922) *Zur Psychologie des produktiven Denkens und des Irrtums*, Friedrich Cohen, Bonn.

Sowa, John F. (1976) "Conceptual graphs for a database interface," *IBM Journal of Research and Development* **20:4**, 336-357.

Sowa, John F. (1984) *Conceptual Structures: Information Processing in Mind and Machine*, Addison-Wesley, Reading, MA.

Sowa, John F. (2000) *Knowledge Representation: Logical, Philosophical, and Computational Foundations*, Brooks/Cole Publishing Co., Pacific Grove, CA.

Sowa, John F. (2003) "Laws, facts, and contexts: Foundations for multimodal reasoning," in *Knowledge Contributors*, edited by V. F. Hendricks, K. F. Jørgensen, and S. A. Pedersen, Kluwer Academic Publishers, Dordrecht, pp. 145-184.

Sowa, John F. (2006) "Worlds, Models, and Descriptions," *Studia Logica*, Special Issue *Ways of Worlds II*, **84:2**, 2006, pp. 323-360.

Sowa, John F., & Arun K. Majumdar (2003) "Analogical reasoning," in A. de Moor, W. Lex, & B. Ganter, eds. (2003) *Conceptual Structures for Knowledge Creation and Communication*, LNAI 2746, Springer-Verlag, Berlin, pp. 16-36.

Stewart, John (1996) *Theorem Proving Using Existential Graphs*, MS Thesis, Computer and Information Science, University of California at Santa Cruz.

Tarski, Alfred (1933) "The concept of truth in formalized languages," in A. Tarski, *Logic, Semantics, Metamathematics*, Second edition, Hackett Publishing Co., Indianapolis, pp. 152-278.

Tesnière, Lucien (1959) *Éléments de Syntaxe structurale*, Librairie C. Klincksieck, Paris. Second edition, 1965.

van Fraassen, Bas C. (1966) "Singular terms, truth-value gaps, and free logic," *Journal of Philosophy* **63**, 481-495.

Way, Eileen C., ed. (1992) *Journal of Experimental and Theoretical Artificial Intelligence* (JETAI), Special Issue on Conceptual Graphs, vol. 4, no. 2.

Whitehead, Alfred North, & Bertrand Russell (1910) *Principia Mathematica*, 2nd edition, Cambridge University Press, Cambridge, 1925.