Implementing a semantic interpreter using conceptual graphs

by John F. Sowa
Eileen C. Way

A parser applies grammar rules to generate a parse tree that shows the syntactic structure of a sentence. This paper describes a semantic interpreter that starts with a parse tree and generates conceptual graphs that represent the meaning of the sentence. To generate conceptual graphs, the interpreter joins *canonical graphs* associated with each word of input. The result is a larger graph that represents the entire sentence. During the interpretation, the parse tree serves as a guide to show how the graphs are joined. Both the front-end parser and the back-end semantic interpreter are written in the Programming Language for Natural Language Processing (PLNLP).

Introduction

When people understand language, they bring to bear a great deal of background knowledge. That knowledge can be organized into four basic categories:

- *Lexical:* Information about word forms.
- *Syntactic:* Information about word and phrase categories and their ordering in sentences.
- *Semantic:* Word definitions, constraints on the use of words in well-formed sentences, and background information about defaults and expectations.
- *Episodic:* Assertions about particular things and events.

Lexical and syntactic information is the easiest to represent and process in a parser. But it is not sufficient to resolve all the ambiguities in natural language; and by its nature, it cannot determine what a sentence means. Semantic information is typically listed in dictionaries, and episodic information is presented in histories, biographies, newspapers, and encyclopedias. Conceptual graphs [1] are used to represent both semantic and episodic information. This paper shows how a semantic interpreter can generate them from a conventional parse tree.

The parser used in this project is the PLNLP English parser developed by Jensen and Heidorn [2]. It uses a machine-readable dictionary of over 70000 words with a grammar that is complete enough to handle almost any English sentence. By a technique of *fitted parsing*, it can even handle ungrammatical sentences, fragments of sentences, and irregularly formed lists and phrases. Yet the parser uses only syntactic rules to generate parse trees. The semantic interpreter translates those trees into conceptual graphs by the following steps:

- For each word of input, it accesses a lexicon of *canonical graphs*, which represent the default ways that concepts and relations are linked together in well-formed sentences.

©**Copyright** 1986 by International Business Machines Corporation. Copying in printed form for private use is permitted without payment of royalty provided that (1) each reproduction is done without alteration and (2) the *Journal* reference and IBM copyright notice are included on the first page. The title and abstract, but no other portions, of this paper may be copied or distributed royalty free without further permission by computer-based and other information-service systems. Permission to *republish* any other portion of this paper must be obtained from the Editor.

```
DECL    NP      NOUN*   "John"
        VERB*   "went"
        PP      PREP    "to"
                NOUN*   "Boston"
        PP      PREP    "by"
                NOUN*   "bus"
        PUNC    "."
```

Figure 1

Parse tree for *John went to Boston by bus.*

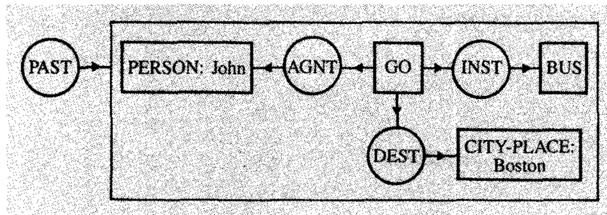

Figure 2

Conceptual graph for *John went to Boston by bus.*

- The interpretation of a complete sentence is formed by joining the small canonical graphs associated with each word to form a large graph that represents the entire sentence.
- The parse trees guide the semantic interpreter by determining the order of doing joins.
- Semantics helps to resolve syntactic ambiguities by rejecting parse trees for which the joins are blocked.

To illustrate this approach, consider the sentence *John went to Boston by bus.* **Figure 1** shows a parse tree generated by the PLNLP English parser for this sentence. DECL indicates that this is a declarative sentence, and the asterisks show the head constituents of each phrase and subphrase. The internal PLNLP records actually contain more detail than Fig. 1 shows: They indicate the past tense of *went*, the singular forms of *John*, *Boston*, and *bus*, and other syntactic and morphological features.

The semantic interpreter generates the conceptual graph in **Figure 2** from the parse tree in Fig. 1. The boxes represent concepts, and the circles represent conceptual relations. Every concept implicitly asserts the existence of something of the corresponding type: This graph asserts the existence of John, a bus, an instance of going, and the city of Boston in the role of place. It further asserts that John is the agent of going, a bus is the instrument, and Boston is the destination of going. To represent the past occurrence of this situation, the monadic relation PAST is attached to a *context* that encloses the entire graph. To save space on the printed page, Fig. 2 can also be written in the *linear form* with square brackets to represent concepts and rounded parentheses to represent conceptual relations:

```
(PAST) → [[GO]-
        (AGNT) → [PERSON: John]
        (DEST) → [CITY-PLACE: Boston]
        (INST) → [BUS]].
```

Besides mapping parse trees to conceptual graphs, the semantic interpreter also checks constraints on well-formed sentences. For the anomalous sentence, *Boston went to birds by spaghetti*, the PLNLP parser does not check any constraints and generates a tree of exactly the same shape as Fig. 1. The semantic interpreter, however, would reject that sentence because the canonical graph for GO requires a MOBILE-ENTITY as agent and a PLACE as destination. This method of allowing the parser to accept anomalous sentences and rejecting them in the semantic stage is fairly common. It contrasts with two other common approaches:

- *Detailed syntactic parsing:* Some parsers require very detailed syntactic features for every word in the lexicon. By using syntactic rules to check the features, they can rule out the possibility of *Boston* as the subject of *went.*
- *Conceptual parsing:* Schankian-style parsers [3, 4] minimize the role of syntax and use the conceptual representation as a guide to selecting words from the input sentence. Some of these parsers may actually override word order and force *birds* to be the agent and *Boston* the destination of *went.*

Splitting the parser and the interpreter simplifies both, while making them more general and easily extendible. Since the PLNLP English parser uses simple features, it can take advantage of conventional machine-readable dictionaries. More detailed parsers, however, require highly complex, specially encoded lexicons; none of them have the range of coverage of the PLNLP parser. Although the semantic interpreter requires a specially encoded lexicon of canonical graphs, they are purely declarative graphs that are easier to generalize than the more procedural code in the Schankian parsers. With a different collection of graphs, the semantic interpreter can be adapted to different domains without any change to the underlying procedures.

Metaphor is another reason for separating the parser and the semantic interpreter. Consider the sentence, *Boston went to the dogs.* Although it is semantically anomalous, it has a metaphorical interpretation that Boston deteriorated in some way. A detailed syntactic parser would reject that sentence completely. A conceptual parser might misinterpret it as meaning that the dogs went to Boston. Yet the PLNLP parser would handle it correctly. After the semantic interpreter failed to generate a conceptual graph for it, the parse tree could be passed to a metaphor interpreter. Although a metaphor interpreter has not yet been written for

this project, this approach allows one to be added as an extension to the normal processing.

Logical level

Conceptual graphs are formally defined, with theorems and proofs that demonstrate the relationships between various aspects. There are two major aspects of the theory that must be considered:

- *Propositions:* The meaning of a declarative sentence is a proposition. For questions, the meaning is a proposition whose truth is to be determined. For commands, the meaning is a proposition stating the result to be achieved. To handle any of these cases, the semantic interpreter must generate a conceptual graph that states a logical proposition.
- *Semantic network:* The background knowledge needed to interpret sentences is more general than any particular sentence. It is organized as a hierarchy of concept types together with their definitions, the constraints on combining them, and the associated defaults and expectations.

The graph for *John went to Boston by bus* is an example of episodic information that asserts a particular proposition. To generate the graph that states that proposition (Fig. 2), the semantic interpreter joins other graphs taken from the semantic network. Conceptual graphs therefore serve two purposes: They may be used by themselves to state propositions; but when stored in the semantic network, they serve as templates or patterns that may be used to generate other graphs.

Before the operations on conceptual graphs can be implemented, each formal object in the theory must be mapped into an appropriate data structure. Following are the basic objects to be represented:

- *Concepts* with type labels and referents.
- *Conceptual relations* with type labels and arcs.
- *Conceptual graphs*, which are interlinked concepts and relations.
- *Contexts*, which are concepts of type PROPOSITION. They support a nesting of conceptual graphs to express negations, modality, and propositional attitudes.
- *Lambda abstractions*, which are conceptual graphs with one or more concepts designated as *formal parameters*.
- A *lexicon* for mapping word forms to their syntactic categories and concept types. (In the book [1], the lexicon is not formally defined, but it is informally represented by the lists in Appendix B.)

Other objects in the theory are specialized uses of these: A canonical graph is simply a conceptual graph that is associated with a concept or relation type; a type definition is a lambda abstraction associated with a type label; and a schema is also a lambda abstraction, but it is used for background information instead of definition.

The semantic network is a repository of general information about some domain of discourse. It must be defined before the semantic interpreter can begin to analyze sentences. During a dialog, the interpreter builds conceptual graphs for new episodic information by using the semantic information as a basis. For every type of concept and relation, there is a node in the semantic network with the following associated information:

- *Type label:* Each type of concept or relation is identified by an uppercase character string, called its type label: CITY and GO represent types of concepts; AGNT and DEST represent types of relations.
- *Type definition:* Some concept and relation types are primitives that cannot be defined. Others are defined by lambda abstractions.
- *Canonical graph:* Every concept and relation type has a conceptual graph that specifies the constraints on the pattern of concepts and relations that may be linked to it. That graph is called its canonical graph.
- *Schema:* A concept type may have one or more schemata that specify defaults, expectations, and other background knowledge. Although there is only one canonical graph for each type, there is no limit to the number of associated schemata.

Canonical graphs show the external pattern of relationships that must be attached to concepts of a given type. They are primarily used in parsing input sentences. Type definitions show the internal pattern of relationships that define a type. They are used in drawing inferences from the input. Schemata may be used for parsing in the same way as canonical graphs, but they are also used for plausible and default reasoning.

The subtype and supertype relations between concept types define a lattice. The pointers that represent the lattice link the types to form the semantic network. Four basic operators are defined on those types:

- *Subtype:* The operator ≤ defines a partial ordering of concept types: PERSON≤ANIMAL; BUS≤MOBILE-ENTITY; GO≤MOVE.
- *Minimal common supertype:* For any types A and B, the type A∪B is the lowest one in the lattice that is above both A and B: PERSON ∪ STONE = ENTITY.
- *Maximal common subtype:* For any types A and B, the type A∩B is the highest one in the lattice that is below both A and B: CITY ∩ PLACE = CITY-PLACE.
- *Conformity:* The operator :: tests whether an individual conforms to a type: CITY::Boston.

These operators are defined theoretically in [1]. In the implementation, they must be defined by procedures that follow pointers up and down the lattice. Other procedures must be implemented for the *canonical formation rules* of copy, restrict, join, and simplify. The derived formation rule of *maximal join* is a combination of the simpler rules that is heavily used by the semantic interpreter. The remainder of this paper shows how the formal objects are mapped into data structures and the formal operators are mapped into procedures.

Implementation level

As a formally defined system, the theory of conceptual graphs could be implemented in LISP, Prolog, or any other programming language. This paper describes a particular implementation in the Programming Language for Natural Language Processing (PLNLP) [5, 6]. PLNLP has facilities for parsing text, generating text, and processing graphs. There were several reasons for choosing it as the implementation language for this project:

- It has a built-in, bottom-up parallel parser, driven by augmented phrase structure grammar rules (APSG).
- The PLNLP English grammar written by Karen Jensen is one of the broadest coverage grammars available for any natural language.
- The PLNLP data structures are specially designed to represent graphs, and the language has a powerful set of operators for building and traversing graphs.

The basic data structure of PLNLP is the *record.* Each record consists of a collection of named attributes with associated values. The values may be simple atoms, or they may be pointers to other records. For a conceptual graph, each node (concept or relation) is represented by a single record. Altogether, eight different kinds of records are used in this implementation:

- Concept records represent the concept nodes (boxes) in a conceptual graph.
- Relation records represent the relation nodes (circles) in a conceptual graph.
- Context records are special cases of concept records whose type is PROPOSITION, but two additional fields are added to speed up certain operations.
- Concept type records are the central directories for semantic information about a concept type. Type and subtype pointers link these records to form the semantic network.
- Relation type records specify semantic information about a conceptual relation type. They are similar to, but slightly different from, the concept type records.
- Lambda records identify the formal parameters of lambda abstractions, which are used in definitions and schemata.
- Lexical records form a dictionary of word forms. Each lexical record contains a list of pointers to word-sense records.
- Word-sense records specify the syntax and semantics for each sense of a word. The semantics is determined by pointers to type records and canonical graphs in the semantic network.

The first attribute of each record is a *tag* that specifies one of these eight kinds of records. The other attributes a record may have are determined by this tag.

PLNLP is a high-level language designed for the kinds of programming problems that occur in natural language processing. PLNLP procedures can be written either as pattern/action production rules or as more traditional sequential programs. A combination of these two forms is used to implement the semantic interpreter algorithms. PLNLP operators support easy record creation, copying, and manipulation. When a new name is encountered in a rule or program statement, that name is automatically defined as a new record attribute. These features result in programs that are shorter and more understandable than LISP code for the same kinds of tasks. Since PLNLP is compiled into LISP, its performance on equivalent operations is the same.

Concepts and relations

Conceptual graphs represent propositions. They may assert episodic information about particular individuals, or they may express general principles in the semantic network. Any representation must satisfy the following constraints:

- *Connectivity:* The algorithms for language parsing, generation, and reasoning depend on the ability to start from any concept and traverse the entire graph. The implementation must support some form of forward and backward pointers linking all the nodes.
- *Generality:* Although most primitive conceptual relations are dyadic, the formalism allows relations with any number of arcs. Furthermore, any concept may have any number of relations attached to it, and the number may increase as more assertions are made. The implementation must support all these options.
- *No privileged nodes:* Any concept in a conceptual graph may be treated as the head. The choice of concept to express as a subject or predicate depends on focus and emphasis, but the representation should not presuppose one choice of root or head (as trees and frames typically do).
- *Canonical formation rules:* The four rules of copy, restrict, join, and simplify are used throughout the system in reasoning and parsing. The implementation must make these operations fast and simple.

Two different record representations were considered for this implementation of conceptual graphs. The first

representation uses separate records for relations and concepts, while the second treats relations as attributes of concept records. The first representation takes more storage space, but it supports greater independence between concepts and relations. The second representation saves some space and may allow a more rapid graph traversal, but it increases the complexity of the record fields and creates difficulties in dealing with relational contraction and expansion. Both representations have merit. The first representation was chosen because it has greater flexibility and simplifies operations on the graphs.

The nodes in conceptual graphs are represented by *concept records* and *relation records*. These represent individual occurrences or tokens of the concept and relation types. A concept record contains the following attributes:

- A *tag field*, indicating that the record is a concept record.
- A *type field*, which points to a concept type record.
- A *referent field*, which specifies the referent for individual concepts or a quantifier for generic concepts.
- A *relation list*, which points to every relation record whose arcs are linked to the current concept.

The referent field of a concept may contain any of the following values:

- A *generic marker*, represented by the * symbol. This marker represents an unspecified individual of the given type.
- An *individual marker*, represented by # followed by an integer identification number.
- A *set referent*, represented by a list of individual markers.
- A *generic set*, represented by the symbol {*}. This indicates that the referent of the concept is a set of zero or more unspecified elements.
- A *quantifier*, represented by a special symbol, such as ∀. This symbol is not one of the primitive forms, but it can be expanded into the primitive forms by the operations defined on conceptual graphs.
- A *definite reference*, represented by # without a trailing integer. This represents an anaphoric reference to be resolved by a coreference link to some other concept.
- A *measure* of some quantity, represented by the marker @. For example, in the concept [SPEED: @55mph], the marker @ shows that 55mph is a measure of the speed, not its name or individual marker.
- A *coreference link*, which connects the current concept to a concept in a dominating context that has the same referent.

Relation records contain a tag field, a type field, and a pointer for each arc. Since every relation record has a pointer to each of the concept records linked to it and each concept record has a list of pointers to the relations linked to it, the resulting conceptual graphs may be traversed in any direction.

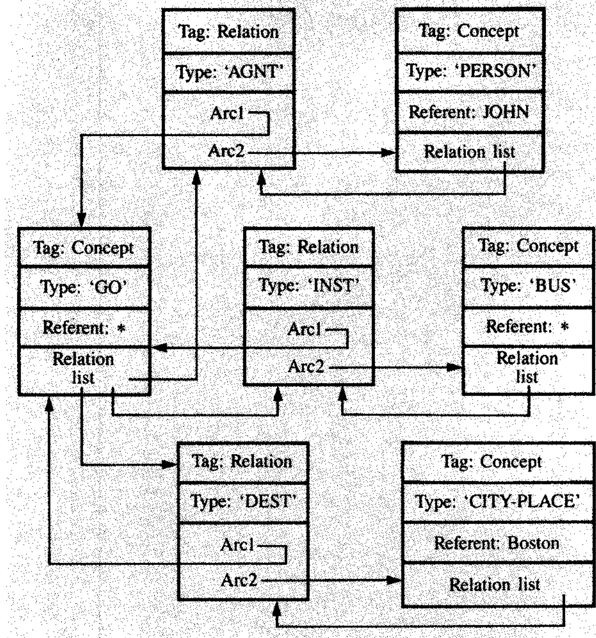

Figure 3

Records that represent the graph in Fig. 2.

In Fig. 2, the conceptual graph for *John went to Boston by bus* is nested inside a context that is marked as past. Since context nodes are not discussed until the next section, **Figure 3** shows only the records for the tenseless graph that is nested inside the context.

Contexts

C. S. Peirce [7] introduced contexts in his existential graphs as a means of grouping propositions. He used them to represent negation, modality, and propositional attitudes. The contexts in conceptual graphs follow Peirce directly, but they are also similar to proposition nodes [8] and partitions [9] in other AI systems. A *context record* is a special case of a concept record of type PROPOSITION. The referent field of a context record contains a list of pointers, each indicating the head of one of the conceptual graphs asserted by that proposition.

Logically, the pointers in the referent field are sufficient to find all the relevant information about a context, and no other fields are needed in a context record. For efficiency, however, two other fields are added:

- The *catalog of individuals* is a list of pointers to all the concepts that are existentially quantified in the current

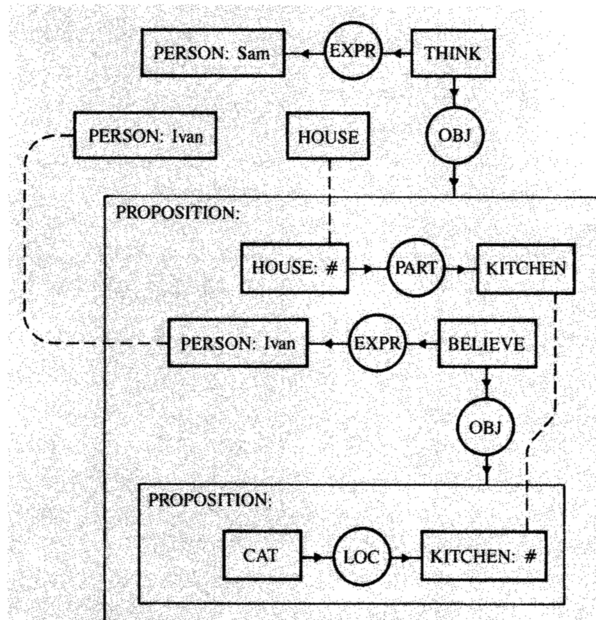

Figure 4

Nested contexts in the logical representation.

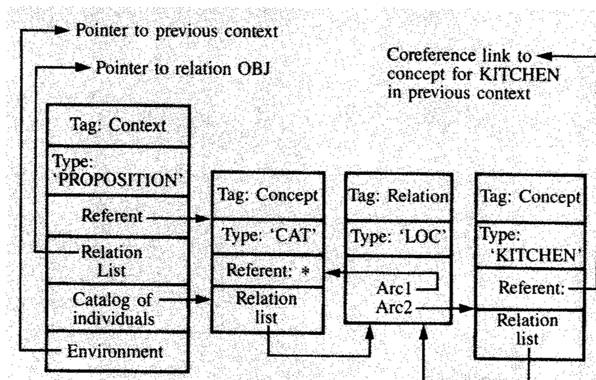

Figure 5

Record representation of the innermost context.

context. This catalog includes concepts of all types: Besides things, it includes events, attributes, and propositions (nested contexts).
- The *environment link* is a pointer to the context in which the current one is nested.

The nesting of contexts is analogous to the nesting of procedures and begin-end blocks in ALGOL-like languages. The catalog of individuals corresponds to the variables that are declared in the current context, and the environment link points to the enclosing context. When resolving pronouns and other anaphoric references, the system first searches the catalog of individuals in the current context. If no referent is found, it follows the environment link to the enclosing context.

To illustrate the nesting of contexts, consider the sentence, *Sam thinks that the house has a kitchen and that Ivan believes that there is a cat in the kitchen.* The graph for this sentence (**Figure 4**) includes three contexts: the outermost context, the context of what Sam thinks, and the context of what Sam thinks Ivan believes. Named individuals, such as Sam and Ivan, are automatically placed in the catalog of individuals for the outermost context (unless it is clear that Ivan exists only in Sam's imagination). Indefinite references, such as *a kitchen* and *a cat*, add new entries to the catalog of individuals for the context in which they occur. Definite references, such as *the house* and *the kitchen*, must be resolved to some individual in the current context or one of its containing environments. Since no referent can be found for *the house*, the system provisionally adds a new concept [HOUSE] to the outermost context and links the inner occurrence of [HOUSE: #] to the outer one. Actions and states, such as thinking and believing, are treated as indefinite references, unless they are described by gerunds with a definite article, such as *the thinking* or *the believing*. As a result of interpreting this sentence, the system constructs three nested contexts with the following catalog of individuals:

- The outermost context has five individuals: Sam, Ivan, a house, Sam's thinking, and a proposition that Sam thinks.
- Sam's thought is a context with three new individuals: a kitchen, Ivan's believing, and a proposition that Ivan believes.
- Ivan's belief is another context with one new individual: a cat.

Figure 4 is shown in the box and circle notation (logical level) rather than the record form (implementation level). A record representation of Fig. 4 would contain 18 records. Since that diagram would be rather complex, **Figure 5** shows only the innermost context (what Sam thinks Ivan believes) expanded into record form.

Lambda abstractions

A lambda abstraction is a conceptual graph with one or more generic concepts identified as formal parameters. Lambda abstractions have multiple uses in the theory:

- *Definitions:* Monadic abstractions are used to define concept types, and n-adic abstractions are used to define n-adic relation types.

- *Single-use types:* Instead of a permanently defined type label, the type field of a concept may contain a lambda abstraction that is created for a single use. These lambda abstractions are commonly created for restrictive relative clauses.
- *Schemata:* Like type definitions, schemata are lambda abstractions. Unlike type definitions, which specify necessary and sufficient conditions, schemata specify defaults and expectations. They are similar in structure, but different in use.
- *Aggregations:* New individuals may be defined as aggregations of parts. These aggregations are typically constructed by specializing the concepts of a lambda abstraction.
- *Prototypes:* A typical individual may be represented by a prototype. It has the same structure as an aggregation, but with average or default values rather than particular values.

To continue the analogy with ALGOL-like languages, a context is like a begin-end block, and a lambda abstraction is like a procedure header.

Implementing lambda abstractions requires a new kind of record, the *lambda record.* It has the following fields:

- A *tag field*, indicating that the record is a lambda record.
- A *parameter count*, which specifies the number of formal parameters.
- A *pointer* for each formal parameter to some generic concept of the conceptual graph that serves as the body of the lambda abstraction.

A lambda abstraction can be used to define the relation *quantity on hand*, with a type label QOH. A database system that has repeated references to part numbers and the quantity of the items in stock may use a relation QOH defined by the abstraction in **Figure 6**.

The conceptual graph in the relational definition serves as the body of the lambda abstraction. The concepts tagged with the variables *x* and *y* are the formal parameters. The relation QOH has two arcs. Its type node has a definition attribute pointing to a lambda record, whose record representation is shown in **Figure 7**.

Consider the sentence, *Every farmer who owns a donkey beats it.* The restrictive relative clause *who owns a donkey* indicates that the quantifier *every* ranges over the donkey-owning farmers. One way to show that is to define a special type DONKEY-FARMER in the semantic network. However, it would be wasteful to clutter up the type hierarchy with a special type for every such clause. Therefore, a single-use lambda abstraction may be defined for this clause. **Figure 8** shows that this lambda abstraction is placed in the type field of the quantified concept. In the record representation, the type field would point to the

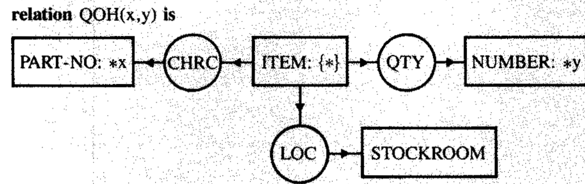

Figure 6

Relational definition for quantity on hand (QOH).

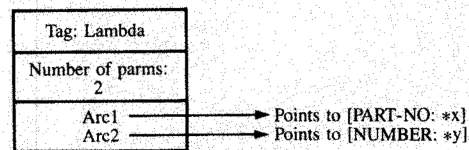

Figure 7

Record representation of the lambda node for QOH.

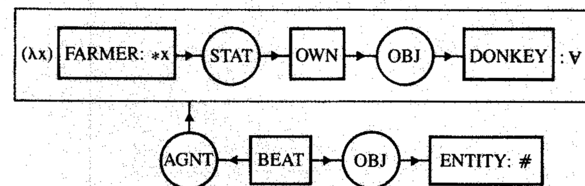

Figure 8

Single-use lambda abstraction for a relative clause.

lambda record instead of a type record in the semantic network.

Figure 8 does not show that the concept [ENTITY:#], which arises from the pronoun *it*, is coreferent with the concept [DONKEY]. Resolution of anaphora proceeds from inside out: A coreference link may only be drawn from a concept with # in its referent field to another concept in the same context or a dominating (enclosing) context. Before the anaphora can be resolved, the universal quantifier ∀ must be expanded into the primitive Peirce form. This expansion is discussed in the section on operations.

Semantic network

The semantic network is represented by a collection of type records for concepts and relations together with the

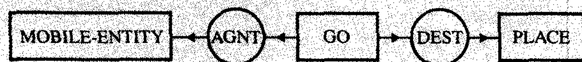

Figure 9

Canonical graph for GO.

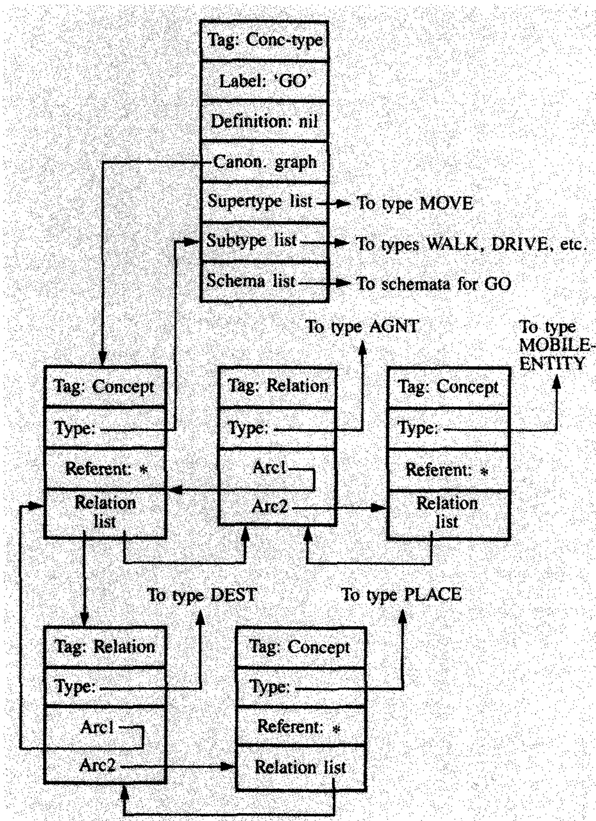

Figure 10

Concept type record and canonical graph for GO.

canonical graphs for each type. Subtype and supertype pointers in the concept type records represent the type lattice. A concept type record has the following fields:

- A *tag field*, indicating a concept type record.
- A *type label*, containing a character string that identifies the concept type.
- A *definition field*, which can either be nil or point to a lambda record for a monadic lambda abstraction.
- A *canonical graph* pointer, which points to the head of the associated canonical graph.
- A *supertype* pointer list, which points to all the supertype records for the given type.
- A *subtype* pointer list, which points to all the subtype records for the given type.
- A *schema* pointer list, which points to the lambda records of all the schemata associated with the given type.

An additional field will be added for prototypes, but they are not yet implemented. **Figure 9** shows a canonical graph for the concept type GO. This graph shows that a MOBILE-ENTITY is the agent of GO and that some PLACE is the destination.

Figure 10 shows the type record for GO with the record form of the canonical graph. This diagram shows explicit pointers from each concept and relation record to the type records. Note that the box and circle notation shows the type labels written inside the nodes. In the record form, the character string form of the type label is written only in the type record. Both representations are consistent with the formal definition, which only says that there must be a function $type(c)$ that maps a concept c into a type label. That function may be supported either by a label in the record or by a pointer to some other record that has the actual label. For efficiency, a pointer is better in the computer implementation: To use the character form to locate the type record would require an associative search or a hash-coded table. Since humans are better at associative searches than at tracing lines on complex diagrams, type labels are better in diagrams designed for people.

Relation type records are not linked in a hierarchy; therefore, they have no supertype and subtype pointer lists. Instead, relation type records have an *arc count field*, which indicates how many arcs are linked to relations of that type. Otherwise, the type record for a relation is similar to that of a concept; both type records have a tag field, a type label field, a canonical graph pointer, and a definition field. If the relation is primitive, then the definition field is nil; but if a new relation type has been defined, then the definition field points to a lambda record. A type record for the primitive relation AGNT is shown in **Figure 11**.

Lexicon

The lexicon maps word forms to syntactic categories and concept types. For each word in the lexicon, there is a *lexical record* that contains a tag field, the word form, and a list of pointers for each word sense. Since each word sense may have a different syntactic category, the *word sense record* must have four fields:

- A *tag* field, indicating that it is a word sense record.
- A *syntax* field, specifying the syntactic category for the word sense.
- A *type* field, pointing to the concept type record for the particular word sense.

- A *head* field, pointing to the concept of the canonical graph for the associated type that serves as head when the concept is expressed by a word of the specified syntactic category. The head concept of a canonical graph is the starting point for doing joins, but the same graph may have different heads when considered from different viewpoints.

Figure 12 shows the lexical record for *hand* with word sense records for two different senses. The first word sense record corresponds to the use of *hand* as a noun referring to the body part (concept type HAND); the second to its use as a verb referring to an act of giving by hand (concept type HAND-GIVE).

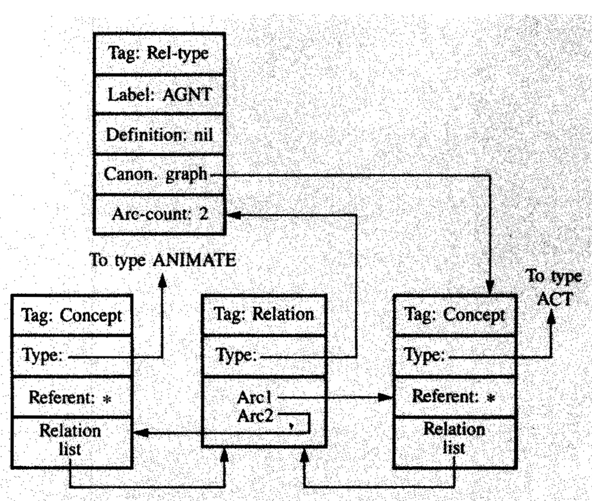

Figure 11

Relation type record with canonical graph for AGNT.

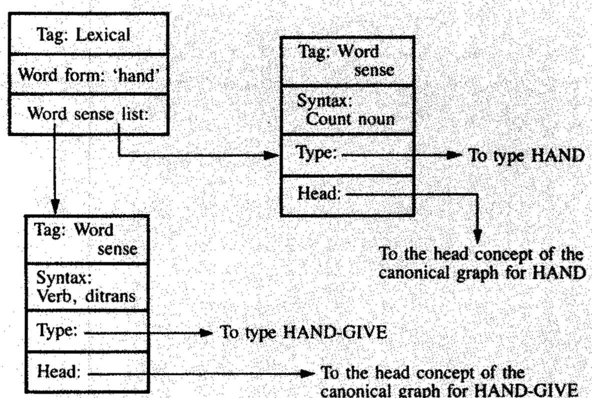

Figure 12

Lexical record and word sense records for 'hand'.

Operations on conceptual graphs

The theory of conceptual graphs includes a notation for knowledge representation and a set of standard operations on that representation. The basic operations include operators on the type hierarchy; the four formation rules of copy, restrict, join, and simplify; and derived formation rules, such as maximal join. These operations are implemented as PLNLP subroutines that are called by the more complex routines for relational expansion, reducing universal quantifiers to primitive form, and anaphora resolution. Following are the basic operations:

- *Lattice operators:* The three operators on the type lattice are *subtype* $\leq$, *minimal common supertype* $\cup$, and *maximal common subtype* $\cap$. For any two type labels A and B, subtype returns *true* if $A \leq B$ and *false* otherwise. The maximal common subtype routine returns a pointer to the type record for $A \cap B$. The minimal common supertype routine returns a pointer to the type record for $A \cup B$. Currently, these operators search the supertype and subtype pointers in the type lattice. An encoding that permits faster searches will be implemented later.
- *Conformity:* For a type A and referent *x*, the conformity routine checks whether *x* conforms to A (written A::*x*). This routine returns the values *true*, *false*, or *permissible*. For example, if Tom is known to be of type MAN, it would return *true* for PERSON::Tom, *false* for WOMAN::Tom, and *permissible* for PEDESTRIAN::Tom.
- *Copy:* The copy routine is a recursive procedure that traverses a graph, creating a new node for each concept and relation it encounters. It is more complex than a tree copy because graph cycles must be considered and the backwards pointers maintained.
- *Restrict:* The restrict routine either replaces the type label of a concept with the label of a subtype or replaces a generic referent with an individual referent. The conformity relation is checked to ensure that the new referent is true or at least permissible for the new type label.
- *Join:* A simple join creates a single graph by merging two graphs on a single matching concept. Given graph A containing concept *x* and graph B containing a matching concept *y*, then the relation list for concept *y* is added to that of *x* and all of the pointers in graph B that point to *y* are reset to point to *x*. Finally, B's concept *y* is erased; all other concept and relation nodes are retained in the combined graph.
- *Simplify:* The simplify routine checks each relation connected to the newly joined concept in order to eliminate any duplicates. Two relation nodes are

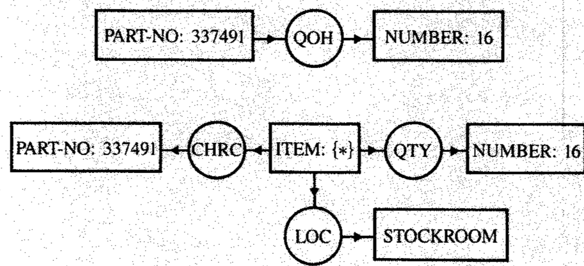

Figure 13

Relation QOH before and after expansion.

considered duplicates when both have the same relation type and both have the same concepts attached to corresponding arcs. If a duplicate relation node is found, it is deleted, and the relation lists in the attached concept nodes are adjusted.

- *Maximal join:* A maximal join is a sequence of joins and simplifications applied to the matching nodes of two graphs. Once the starting place for the maximal join is determined, a simple join is performed on the matching concepts. Next, all the nodes adjacent to the joined concept are checked to see if any of the relations from one graph match those of the other. If a match is found, then the procedure continues around the graph locating matching concepts, restricting their types, joining and simplifying until no further matches are detected.

Other operations on conceptual graphs are *relational expansion* and *contraction.* Relational expansion replaces a relation and its attached concepts with the expanded form of its relational definition. In the section on lambda abstractions, a new relation QOH was defined. In the type node for that relation, the definition field points to the lambda record shown in Fig. 7. The expansion operation copies the conceptual graph designated by the lambda record, joins the concepts attached to the QOH relation with the formal parameters of the lambda record, and deletes the original QOH relation record. **Figure 13** shows that relation before and after expansion.

The relational contraction operation, while not in itself difficult, requires complex pattern matching to determine which subgraph is a candidate for contraction. Since the contraction operation is not important for semantic interpretation, it has not yet been implemented.

Another operation on conceptual graphs is the expansion of universal quantifiers into Peirce's primitive existential form. This operation is illustrated for the sentence, *Every farmer who owns a donkey beats it,* whose conceptual graph was shown in Fig. 8. The result of expanding the universal quantifier in that graph is shown in **Figure 14**. The expansion takes place according to the following steps (page numbers refer to the book [1]).

1. Draw a double negation around the entire graph in Fig. 8. This step is always permitted by the rules of inference for conceptual graphs (Assumption 4.3.5, p. 154).
2. Expand the universal quantifier according to its definition (Assumption 4.2.7, p. 146):
   a. Make a copy of the concept with the universal quantifier (the one that represents *every farmer who owns a donkey*), and place it between the inner and outer negative contexts.
   b. Draw a coreference link from the universally quantified concept in the inner context to its copy in the enclosing context.
   c. Erase the universal quantifier both on the original concept and its copy.
3. By Theorem 4.3.7 (p. 158), any concept type in an evenly enclosed context may be generalized to a supertype: The innermost lambda abstraction for donkey-owning farmers may be simplified to just the type label FARMER.
4. Since the universal quantifier has been removed from the outer copy, it is possible to expand the lambda abstraction by a maximal type expansion (Definition 3.6.7, p. 109).

After this expansion has been done, the anaphora can be resolved from the innermost context outward to generate Fig. 14. This method of resolving references follows the accessibility constraints of discourse representation theory [10]. Such constraints are not always sufficient to determine the correct referent, and semantic and pragmatic constraints must also be used. Those constraints have not yet been implemented, but the current system should provide a useful tool for exploring various techniques.

Syntax-directed generation of conceptual graphs

The semantic interpreter starts with the parse tree produced by the PLNLP English grammar. It determines the order of joining canonical graphs associated with each input word. Three attributes of the parse records are especially important for traversing the tree: the head, the premodifiers, and the postmodifiers. The head attribute points to a record for the head of a phrase. The head is determined by purely syntactic criteria. The premodifier list has a pointer to a record for each premodifier, and the postmodifier list has a pointer to a record for each postmodifier.

The head, premodifier, and postmodifier attributes do not occur in the terminal records for the input words, but they do occur in the records for all other subtrees of a parse tree. Therefore, the conceptual graph for a sentence can be generated by a recursive algorithm: If a record has no head

attribute, then the record is a terminal node, and the canonical graph for that word is returned; otherwise, the conceptual graph is formed by a maximal join of the graph for the head with the conceptual graphs of the premodifiers and postmodifiers.

The starting positions for joins are usually determined by syntactic criteria. The following rule, for example, shows a verb phrase VP formed from a verb V with a prepositional phrase PP as postmodifier:

VP → V PP

For the canonical graph for V, the head is the concept associated with the verb. For the canonical graph for PP, the head is a concept that represents the verb to be modified. For the canonical graph for VP, the head is the result of joining the heads of the V and PP graphs. In general, syntactic criteria determine the starting points for joining the main modifiers in English: adjectives modifying nouns, adverbs modifying verbs, and prepositional phrases modifying either nouns or verbs. Not all starting points are defined so clearly, however. For joining the subject to the main verb, the interpreter tries a list of preferences in Fillmore's order, AGNT, INST, OBJ [11]. For nouns modifying other nouns—one of the most ambiguous aspects of English—the program compares the head concept of the modifier graph with all the concepts in the canonical graph for the principal noun. If no join is possible, then the interpreter reports a failure to determine how the modifier and principal noun are related.

Consider the parse tree in Fig. 1 for the sentence *John went to Boston by bus.* The interpreter starts by finding the canonical graphs for each terminal node (word) of the tree. To generate the graph for the first PP node, it joins the graph for the preposition *to* with the graph for *Boston.* The resulting graph is joined to the graph for the main head *went.* Next, the graph for the preposition *by* is joined with the graph for *bus*, and this graph is joined with the previously joined graph for *went* and the first PP. Finally, the graph for the premodifier *John* is joined to generate the graph for the entire sentence. Generating a conceptual graph is not always so straightforward. Two basic kinds of ambiguities may arise:

- *Lexical ambiguity:* Words may have multiple senses with different canonical graphs. The preposition *to*, for example, may indicate the destination (DEST) or recipient (RCPT), and *by* may indicate instrument (INST), location (LOC), or agent (AGNT).
- *Structural ambiguity:* The point of attachment for subtrees of the parse tree may not be uniquely determined by the grammar. In the sentence *John went to the chair by the window*, the phrase *by the window* is a postmodifier for *the chair*, yet the first parsing shows it as a postmodifier for the verb.

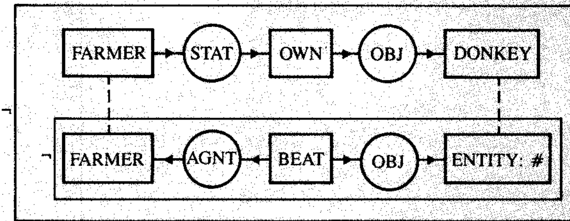

Figure 14

Expanded form of Fig. 8.

To handle lexical ambiguities, the semantic interpreter must consider multiple candidate graphs and pass them up the parse tree until one or more are blocked by a failure to find an acceptable join. Therefore, at each level of the parse tree, there may be a list of several graphs for different interpretations of the lower branches. The program tries to join each combination of graphs from the given lists by sending the maximal join procedure two graphs at a time. The successful joins are evaluated by counting the number of concepts in the resulting graphs. The graph with the smallest number of concept nodes is preferred, since that is the one with the largest number of matching concepts (a maximal join). The preference for maximal joins is similar to Wilks's method of *preference semantics* [12].

To handle structural ambiguities, Heidorn and Jensen [13] developed a method of moving nodes in the parse tree. The parser first generates exactly the same structure for *John went to Boston by bus* and *John went to the chair by the window.* To support node moving, however, the parser keeps a list of other possible attachments for the modifiers. When joining graphs, the semantic interpreter would find that *the window* is not an acceptable instrument for *went.* The node-moving technique would try another option of putting the second PP in the postmodifier list for *the chair.* Then the semantic interpreter would find that *the window* is an acceptable location for *the chair* and join the canonical graph for *by* indicating the LOC relation. The technique of generating a single parse tree and adjusting it by moving nodes is more efficient than generating all possible trees and throwing away ones that violate the constraints.

The interpreter described in this paper uses only canonical graphs to determine the connections between the input concepts. Schemata for a type are typically larger than the canonical graphs for that type: They include more background knowledge and a more extensive pattern of relationships. One of the extensions to be explored is the use of schemata as an adjunct to the canonical graphs in semantic interpretation. There are two possible ways of using them:

- If an ambiguity cannot be resolved by canonical graphs alone, try possible joins with schemata and take the result that has the largest number of matching concepts.
- Since a schema has a large pattern of relationships, any schema that is applicable could resolve many ambiguities at once. Therefore, it might be more efficient to try joining schemata before or instead of the simpler canonical graphs.

Either of these approaches could be implemented as an extension to the current interpreter. The basic algorithms would remain unchanged, and the only difference would be the use of schemata instead of or in addition to the canonical graphs. But since the number of schemata may be much larger than the number of canonical graphs, the efficient use of schemata requires an associative search or preference strategy for finding the most likely candidates to try.

Other implementations

All implementations of conceptual graphs that conform to the formal definition [1] must, at the logical level, be isomorphic. Because of the isomorphism, it is possible to write conversion routines that map the data structures from one version to another. For example, output from the semantic interpreter could be sent to a formatter that displays the graphs as boxes and circles on a screen, to a theorem prover that does inferences from them, to a database system that stores and retrieves them, or to a language generator that translates them into some other language (natural or artificial). Even if those other systems used a different internal representation, they could express the same information at a logical level.

Besides the implementation described in this paper, processors for conceptual graphs are being implemented at several locations in IBM and at universities:

- The KALIPSOS Project at the IBM Paris Scientific Center is using conceptual graphs for a knowledge acquisition system [14]. They are using Prolog to develop a parser for French and an inference engine that processes Prolog-like rules with the predicates represented as conceptual graphs. They have also implemented a variety of tools for helping a knowledge engineer to analyze natural language text in order to define the rules and facts of a knowledge base.
- The Intelligent Help Project at the IBM Los Angeles Scientific Center is developing a computer help system based on conceptual graphs [15]. They have been analyzing typical help requests to determine how they could be represented and processed with conceptual graphs and have developed a standard interchange notation for mapping conceptual graphs from one system to another. Using that notation, they have developed a processor for displaying the graphs on screens and printers in the box and circle form as well as printing them in the linear form.
- At the University of Bristol, Morton and Baldwin [16] have used Prolog to implement a conceptual graph processor with extensions to handle fuzzy referents. Their front-end parser handles elliptical queries, anaphoric references, user definitions, and meta-queries. The back-end maps conceptual graphs generated by the parser into FRIL, a fuzzy relational database query language. They are also using conceptual graphs to represent spatial and graphic relationships for a computer vision system.
- At Deakin University, Garner and Tsui [17] are representing audit information in conceptual graphs. They have built a knowledge acquisition facility that enables an expert to define concept types with associated canonical graphs for any application. They also implemented an inference engine that does frame-like reasoning with conceptual graphs. Their English front-end just uses a simple template pattern matcher, but they plan to replace it with a more general parser.
- At the IBM Japan Science Institute, Maruyama [18] has used Prolog to implement actors attached to conceptual graphs (as described in [1], Section 4.6). Starting with a query graph for a user's question, the system joins schemata containing attached actors. Control marks on the graphs trigger the actors to access database relations or do computations. The result of satisfying the control marks is the answer to the original query. For answering typical database queries, actors attached to conceptual graphs appear to be more efficient than a general inference engine. Maruyama has also written a translator for mapping propositions stated as conceptual graphs into Prolog clauses.
- At the IBM Toronto Laboratory, the Machine-Readable Information Project has implemented a parser and semantic interpreter in Prolog [19]. The grammar was mapped into Prolog from the context-free rules of the Linguistic String Project [20]. But instead of using the LSP restriction rules, they let the semantic interpreter use canonical graphs to check constraints on the parsing. They have also implemented a graphics editor for defining and displaying conceptual graphs.

Since Prolog supports different kinds of data structures from PLNLP or LISP, the Prolog implementations must use different encodings for the same logical information. For the sentence, *Felix the cat is chasing a mouse*, the conceptual graph in linear form would be

[CHASE]–
  (AGNT) → [CAT: Felix]
  (OBJ) → [MOUSE].

One method of representing such a graph is to assign a unique identifier to each concept (c1, c2, and c3) and represent each relation by a predicate. That graph could then be represented by five Prolog assertions:

```
id(c1, cat.felix).          agnt(c2,c1).
id(c2, chase.'*').          obj(c2,c3).
id(c3, mouse.'*').
```

An assertion like id(c2,chase.'*') means that c2 identifies a concept whose type label is CHASE and whose referent is * (a generic concept). The assertion agnt(c2,c1) means that concept c2 has an agent c1. Yet this representation is too limited: All concepts are at the same level, and there is no way to show the nesting of contexts. A more general representation is to show a conceptual graph as a list of concepts with unique identifiers followed by a list of relations:

```
cg((c1.cat.felix).(c2.chase.'*').(c3.mouse.'*').nil,
   (agnt.c2.c1).(obj.c2.c3).nil).
```

The dyadic function cg identifies the concept and relation lists of a conceptual graph. This structure could then be nested inside the referent of a concept of type PROPOSITION (a context). For graph traversals, character string identifiers like c1 and c2 are less efficient than the direct pointers in PLNLP (and its underlying LISP system). But the backtracking and unification algorithms in Prolog may simplify other operations. In any case, the various processors implemented in both languages are quite fast, even though they are still experimental tools.

Acknowledgments

Both the design and implementation work on this project benefited from collaboration with the PLNLP group, especially George Heidorn, Karen Jensen, and Norman Haas. The paper itself benefited from comments and suggestions by George Heidorn, Alex Hurwitz, Hiroshi Maruyama, and the referees.

References

1. J. F. Sowa, *Conceptual Structures: Information Processing in Mind and Machine*, Addison-Wesley Publishing Co., Reading, MA, 1984.
2. K. Jensen and G. E. Heidorn, "The Fitted Parse: 100% Parsing Capability in a Syntactic Grammar of English," *Proceedings of the Conference on Applied Natural Language Processing*, Santa Monica, CA, Association for Computational Linguistics, 1983, pp. 93–98.
3. *Conceptual Information Processing*, R. C. Schank, Ed., North-Holland Publishing Co., Amsterdam, 1975.
4. *Inside Computer Understanding: Five Programs Plus Miniatures*, R. C. Schank and C. K. Riesbeck, Eds., Lawrence Erlbaum Associates, Hillsdale, NJ, 1981.
5. G. E. Heidorn, "Natural Language Inputs to a Simulation Programming System," *Report NPS-55HD72101A*, Naval Postgraduate School, Monterey, CA, 1972.
6. G. E. Heidorn, "Augmented Phrase Structure Grammar," *Theoretical Issues in Natural Language Processing*, R. C. Schank and B. L. Nash-Webber, Eds., Association for Computational Linguistics, 1975, pp. 1–5.
7. C. S. Peirce, manuscripts on existential graphs, reprinted in *Collected Papers of Charles Sanders Peirce*, A. W. Burks, Ed., Vol. 4, pp. 320–410, Harvard University Press, Cambridge, MA. For a summary of these graphs, see D. D. Roberts, *The Existential Graphs of Charles S. Peirce*, Mouton, The Hague, 1973.
8. S. C. Shapiro, "The SNePS Semantic Network Processing System," *Associative Networks: Representation and Use of Knowledge by Computers*, N. V. Findler, Ed., Academic Press, Inc., New York, 1979, pp. 179–203.
9. G. G. Hendrix, "Expanding the Utility of Semantic Networks Through Partitioning," *Proc IJCAI-75*, pp. 115–121 (1975).
10. H. Kamp, "Events, Discourse Representations, and Temporal Reference," *Langages* **64**, 39–64 (1981).
11. C. J. Fillmore, "The Case for Case," *Universals in Linguistic Theory*, E. Bach and R. T. Harms, Eds., Holt, Rinehart and Winston, New York, 1968, pp. 1–88.
12. Y. A. Wilks, "An Intelligent Analyzer and Understander of English," *Commun. ACM* **18**, 264–274 (1975).
13. G. E. Heidorn and K. Jensen, "Parsing by Building and Adjusting an Approximate Parse Tree," presented at Workshop on Semantics and Representation of Knowledge, New York, 1983.
14. Jean Fargues, Marie-Claude Landau, Anne Dugourd, and Laurent Catach, "Conceptual Graphs for Semantics and Knowledge Processing," *IBM J. Res. Develop.* **30**, No. 1, 70–79 (1986, this issue).
15. A. Hurwitz, IBM Los Angeles Scientific Center, personal communication.
16. S. K. Morton and J. F. Baldwin, "Conceptual Graphs and Fuzzy Qualifiers in Natural Language Interfaces," presented at the Cambridge Conference on Fuzzy Sets, 1985. To appear in *Fuzzy Sets and Systems*, 1986.
17. B. J. Garner and E. Tsui, "Knowledge Representation in the Audit Office," *Australian Comput. J.* **17** (August 1985).
18. H. Maruyama, "Towards a Discourse Analysis Using Conceptual Graphs," presented at a working group meeting of the Information Processing Society of Japan, 1985 (written in Japanese).
19. J. Wilson and C. Tandy, IBM Toronto Laboratory, personal communication.
20. N. Sager, *Natural Language Information Processing*, Addison-Wesley Publishing Co., Reading, MA, 1981.

Received August 12, 1985; revised September 1, 1985

John F. Sowa *IBM Systems Research Institute, 500 Columbus Avenue, Thornwood, New York 10594.* Mr. Sowa is a senior staff member at the IBM Systems Research Institute. After receiving a B.S. in mathematics from the Massachusetts Institute of Technology, Cambridge, in 1962, he joined an applied mathematics group at IBM. Four years later, he attended graduate school at Harvard University, earning an M.A. in applied mathematics under the IBM Resident Graduate Study Program. At IBM, his early work was in programming languages and machine architecture. Since 1972, he has concentrated on artificial intelligence and natural language processing. His research has appeared in numerous articles and in his book, *Conceptual Structures: Information Processing in Mind and Machine*, which was published by Addison-Wesley. Mr. Sowa belongs to a number of professional societies, including IFIP Working Group 2.6, which is concerned with knowledge representation in databases and AI.

Eileen Cornell Way *State University of New York, Department of System Science, Binghamton, New York 13901.* Ms. Way is a graduate student at the State University of New York. She has received a B.S. in philosophy from Harpur College, Binghamton, New York, and an M.S. from the School of Advanced Technology at the State University of New York at Binghamton. She implemented the semantic interpreter described in this paper while collaborating with John Sowa at the IBM Systems Research Institute. Ms. Way plans to continue developing and extending the interpreter as part of her work for her Ph.D. dissertation. Her research interests include artificial intelligence, natural language processing, and knowledge representation. She is a member of the Association for Computing Machinery, the Society for General Systems Research, and the American Philosophical Association.