Representation of Knowledge using Sowa's Conceptual Graphs: An Implementation of a Set of Tools

by

Gary Schiltz

B.S., Kansas State University, 1981

A MASTER'S REPORT

Submitted in Partial Fulfillment of the

Requirements for the Degree

MASTER OF SCIENCE

Department of Computer Science

KANSAS STATE UNIVERSITY
Manhattan, Kansas

1986

Approved by:

Major Professor

Committee Member

Committee Member

TABLE OF CONTENTS

CHAPTER 1 : INTRODUCTION

Traditionally, computer systems have been very limited in the range of problems that they are capable of solving. In the early days of computing, it was assumed that it would be a relatively straightforward task to make computers as powerful and flexible as the human mind. Presumably, the most important element lacking was bigger and more powerful computer hardware.

However, in our failure to make truly intelligent computer systems, we have found that bigger and better hardware alone does not help us produce intelligent, autonomous computer systems. Tasks which humans find quite easy to accomplish, such as reading and understanding, are in fact so inherently difficult that we have very little idea how to make computer systems capable of performing them. Apparently, in order to make intelligent computers, we must first understand the nature of intelligence.

1.0 Knowledge and Intelligence

Central to understanding intelligence is the issue of knowledge and its representation. According to Rich (1983), "One of the few hard and fast results to come out of the first 20 years of A.I. research is that intelligence requires knowledge." And Winston (1984), maintains that "...the inherent thinking advantages of powerful representations enable progress that would be impossibly difficult with anything less adequate." So, from that point of view, progress in making intelligent computers depends largely upon our understanding of knowledge, and our ability to adequately represent it.

1.1 Traditional Views of Knowledge

The question of what it means to know something is a central theme in the field of philosophy. Although the search for the essence of knowledge has occupied philosophers throughout history, their theories have been largely untestable, so few definite conclusions have been made.

More recently, psychologists have attempted to define knowledge in terms of the human mind. The most current psychological school of thought, known as cognitive psychology, proposes that knowledge of something is the ability to form mental models of that thing. Here, intelligence is the ability to form mental models and to do processing which is guided by those models.

Computer scientists in the area of database theory attempt to create models which represent parts of the real world through schematic descriptions. They seek to create

these models with constraints which will maintain the models as faithful representations of the external reality being modeled. However, the descriptions are generally very limited, dealing with a quite specific area of external reality. And, the constraints imposed must be specified for each model; i.e., the database system itself has no knowledge about the real world, other than what is specified in the constraints of the model.

1.2 Artificial Intelligence and Knowledge

The field of artificial intelligence has the potential of bringing together theories from these seemingly distinct fields. By creating computer models of theories in cognitive psychology, we gain insight into the nature of human intelligence by having a controlled medium on which to test the theories. This insight can shed light on the essence of knowledge and meaning, and thus is of value to philosophers, prompting further philosophical questions to be asked.

Almost as a by-product of this search for the essence of meaning through computer models of cognitive behavior, will come the production of highly flexible and user-friendly computer systems. These systems will have real-world knowledge incorporated into them.

In fact, there are many who believe that computers can be made to demonstrate generally intelligent behavior. Newell and Simon's [Newell, 1976] Physical Symbol System Hypothesis states that a system of physical symbols "has the necessary and sufficient means for general intelligent action."

Sowa (1984) has written extensively on the subject of the nature of intelligence in his book "Conceptual Structures: Information Processing in Mind and Machine", and proposes a theory for the representation of knowledge to facilitate the study of the nature of intelligence. The author believes that the theory of conceptual graphs as presented by Sowa is the most promising tool for exploring the nature of generally intelligent behavior.

Sowa also informally presents a linear notation as the vehicle of this representation. He does not, however present a formal specification of this representational notation.

Furthermore, the author has found no reference to the existence in public domain of any software for parsing of this linear representational notation. The author also believes that such software would be an asset to anyone wishing to test the theories presented in Sowa's book. In the period from June, 1985 through November, 1985, the author has specified a grammar to formally describe this notation, and has implemented in Lisp a parser as a starting

point in the implementation of a system for representing knowledge using this linear notation.

This paper serves as an introduction to knowledge representation, introduces Conceptual Graph theory and notation as presented by Sowa, and outlines both the tools which the author believes need to be implemented, as well as the tools actually implemented by the author.

CHAPTER 2 : CONCEPTUAL GRAPH SUMMARY

The theory of conceptual graphs is a knowledge representation theory formulated by Sowa (1984), which is described in full in his book "Conceptual Structures: Information Processing in Mind and Machine". Although many of the ideas which make up the theory are not new, his would seem to be the most complete and testable theory of knowledge representation.

This section serves two purposes. First, through examples and informal descriptions, the linear notation for representing conceptual graphs is presented. Second, through this same presentation of examples, the theoretical aspects of conceptual graphs are introduced.

2.0 Basic Notation

A conceptual graph has the form of a connected, bipartite graph. The two kinds of nodes in each graph are concept nodes, which represent concepts such as entities, objects, ideas, states and events; and conceptual relation nodes, which indicate how the concepts are related to each other. Each conceptual relation has one or more arcs, each of which must be linked to some concept.

In the linear notation, concepts are enclosed in square brackets, conceptual relations are enclosed in parentheses, and arcs are represented by right or left arrows. The end of the graph is indicated by a period. For example, to represent the idea of "A cat sitting on a mat", we could write:

[CAT] <- (AGNT) <- [SIT] -> (LOC) -> [MAT].

This is meant to represent some act of SITting, in which the LOCation of the act is on a MAT. and the AGeNT of the sitting is a CAT. The concept SIT, in this graph, is linked to two other concepts (CAT and MAT), through two conceptual relations.

Note that if a concept is linked to more than two concepts, we cannot represent this with only the right and left arrow notation, since in a linear form, only two directions exist (ahead of the current position, and behind the current position). So, we use the notation of a minus sign after a concept to indicate that an unspecified number of arcs are to follow. To specify the end of a list of arcs, the minus sign is matched by a comma, unless the end of the graph has been reached, in which case the period matches with all unmatched minus signs. So, to represent "A cat sitting placidly on a mat yesterday" (SIT is linked to four other concepts), we would write:

```
[SIT]-
   (AGNT) -> [CAT]
   (LOC)  -> [MAT]
   (MANR) -> [PLACIDLY]
   (PTIM) -> [YESTERDAY].
```

This represents some act of SITting, in which the AGeNT of the act is a CAT. the LOCation of the act is a MAT, the MANneR of the sitting is PLACIDLY, and the Point in TIMe in which the act occurred is YESTERDAY.

2.1 Labels and Meaning

It is important to point out that the concept and conceptual relation labels in themselves have absolutely no meaning. The label CAT, from the previous example, doesn't necessarily have anything to do with a furry, four-legged mammal commonly kept as a pet. It could just as well refer to a machine used to wash clothes in. Conversely, the concept of a cat could be represented just as well by the label CONCEPT11304, or any other label. An important point in conceptual graph theory is that concepts and conceptual relations exist independently of their representation. Labels are merely handles through which we can access information about a concept. Where meaning is located in conceptual graph theory is the subject of sections 2.3 through 2.8.

2.2 Generic Concepts and Referents

Concepts such as those previously used (e.g. those referred to by the labels CAT and MAT) are known as generic concepts. They represent an unspecified individual of the kind represented by their concept label. In this sense, they are like variables with no current value. For example, CAT stands for an unspecified individual instance of the concept pointed to by the label CAT. It doesn't refer to any particular CAT or CATs.

However, in the real world, there are individual instances of concepts, as well as abstract concepts themselves. To deal with this fact, the idea of a referent is introduced. In simple terms, a generic concept label refers to the type of concept being represented, while a referent refers to an individual. A referent is separated from the concept type label by a colon; for example, to represent the concept "Rosie the cat sitting on a mat", we would write:

```
[SIT]-
   (AGNT) -> [CAT: Rosie]
   (LOC)  -> [MAT].
```

This can be thought of as an act of SITting in which the AGeNT of the act is an individual of type CAT (referenced by the label Rosie) and the LOCation of the act being on a MAT. Note that the label Rosie actually has no meaning in itself; it simply points to an individual of type CAT. In section 2.5, we will look at how individuals are actually defined.

Note that even a generic concept has an implied referent, that of an unspecified individual, which corresponds to a variable with no current value. Although it is normally left off, the referent of a generic concept is the asterisk. So, the concepts [CAT] and [CAT: *] are equivalent and mean an unspecified individual of type CAT.

Also, just as variables in some programming languages can have different types of values (e.g. integers, pointers, sets, etc.), referents can also have different forms. These forms include constants (individual markers, conceptual graphs, measures), variables and different types of sets of constants and variables.

2.3 The Nature of Meaning

Before continuing with more advanced ideas in conceptual graph theory, it is important to first understand the nature of the meaning of something. As pointed out in section 2.1, there is no inherent meaning in the concept and conceptual relation labels.

Where, then does the meaning of a concept or group of concepts lie? In conceptual graph theory, meaning can be thought of as the way in which concepts are related to each other. Sowa uses the term semantic network to refer to the collection of all relationships that concepts have to each other. He points out that in philosophy, White (1975) considered the meaning of a concept to be the position of the concept in this vast network. According to White, "Just as the identity of a point is given by its coordinates, ... so the identity of a concept is given by its position relative to other concepts. ... A concept is that which is logically related to others just as a point is that which is spatially related to others."

2.4 Definition of Types

Aristotle believed that all things could be defined in terms of other things, introducing the notion of genus and differentia. The genus of a concept can be thought of as its supertype, and the differentia specifies how the concept is different from its supertype. For example, the concept of an AIRPLANE might be represented in terms of the concepts FLYING and VEHICLE. FLYING is the differentia that

distinguishes an AIRPLANE from all other VEHICLEs. Essentially, each concept fits somewhere in a type hierarchy.

Sowa has expanded the notion of a genus and differentia through the use of the type definition. A type definition is basically of the form

```
type <type name> ( <formal parameter> ) is
   <conceptual graph>
```

The formal parameter must appear in the conceptual graph as the referent of the concept which is the supertype of the type being defined. For example, to define a type AI-TEACHER, we show how this type relates to (differs from) a type already in the hierarchy (say, TEACHER):

```
type AI-TEACHER (x) is
   [TEACHER: *x]-
      (TEACH) -> [SUBJECT: AI]
      (LOC)   -> [UNIVERSITY].
```

This says that an AI-TEACHER is some TEACHER who TEACHes the subject AI (the asterisk in front of the variable x in the graph means that the value of x should serve as the referent).

A type definition is much like a dictionary definition. Each word comprising a dictionary definition is, itself, in the dictionary, where it is also defined in terms of other words. A dictionary is in fact a linguistic model of the known world. Accordingly, the type hierarchy is a conceptual model of the known world.

Also note that the type hierarchy is considered to be a necessary and sufficient means of classifying an individual (necessary means that all the information contained in the type hierarchy is required to distinguish individuals one from another; and, sufficient means that no information other than what is contained in the hierarchy is required in order to distinguish individuals one from another).

This view of the world as being 100 percent classifiable in terms of a type hierarchy (genus and differentiae) is often referred to as an Aristotelian view of the world, since Aristotle was the first to propose the view. There is no way in this view to account for uncertainty in classification. For example, there is no room for saying that an individual "would seem to be" or "most likely is" of a particular kind. However, the human view of the world is full of such uncertainty. While the Aristotelian model of the world is deterministic, humans would seem to create stochastic models of the world. Sections 2.7 and 2.8 address the issue of how to handle knowledge which doesn't seem to fit in the Aristotelian model.

2.5 Definition of Individuals

The previous section introduced the idea of defining types in terms of how they relate to (or are different from) other types. Individuals are defined in much the same way. To define an individual of some type, it is necessary to specify how that individual is different from other individuals of that type. Since an individual is a specific instance of a type, an individual is defined by making specific (adding a non-generic referent to) one or more concepts which are generic in the base type. The syntax of an individual definition is similar to that of a type definition:

```
individual <type name> ( <individual label> ) is
   <conceptual graph>
```

The individual label serves as a pointer to the individual definition, and, just as with the formal parameter of a type definition, this label must appear in the conceptual graph of the definition. For example, to show that Joe Schlunk is an AI teacher who teaches at KSU, we would say:

```
individual AI-TEACHER (Joe Schlunk) is
   [TEACHER: Joe Schlunk]-
      (TEACH) -> [SUBJECT: AI]
      (LOC)   -> [UNIVERSITY: KSU].
```

Before a label can be used as the referent of a concept, it must previously have been defined as an individual (except in the graph in which it is being defined). For example, the label "Joe Schlunk" cannot be used as a referent until after the above individual definition has been made.

2.5.1 The Conformity Relation

In conventional programming languages, the value of a variable must conform to the type of the variable. Similarly. the referent of a concept must conform to the type of the base type, or to any of its supertypes. If it does, then the conformity relation is said to hold. For example, assume the individual definitions have been made (where "..." means the rest of the definition is unimportant in this discussion):

```
individual BEAGLE (Snoopy) is
   [DOG: Snoopy]-
   ...

individual COLLIE (Lassie) is
   [DOG: Lassie]-
   ...
```

The concepts [COLLIE: Lassie] and [BEAGLE: Snoopy] are valid (the referents 'conform' to the base types), whereas [COLLIE: Snoopy] and [BEAGLE: Lassie] violate the conformity relation (the referents do not conform to the base types). Also, assuming that ANIMAL is a supertype of DOG (and therefore a supertype of BEAGLE and COLLIE), the conformity relation holds for the concepts [ANIMAL: Lassie] and [ANIMAL:Snoopy].

2.6 Definition of Conceptual Relations

Up to now, we've used conceptual relations without any notion of how those relations are defined. Just as concepts are defined in terms of other concepts and conceptual relations, conceptual relations are defined in terms of concepts and other conceptual relations. The syntax of a conceptual relation definition is similar to that of a type definition:

```
relation <relation name> ( <formal parameter> ) is
   <conceptual graph>
```

The formal parameter must appear in the conceptual graph as the referent of some concept. For example, to define the monadic relation FUTURE, we could say

```
relation FUTURE (x) is
   [SITUATION: *x]-
      (PTIM) -> [TIME: *] -> (>) -> [TIME: Now].
```

This says that the relation FUTURE modifies a SITUATION, the Point in TIMe of which is a TIME which is greater than the TIME called Now (it is assumed that the value of Now varies, depending upon when the graph using the relation is created). FUTURE can modify concepts of type SITUATION, or concepts which are subtypes of SITUATION.

2.6.1 The Canonical Base

The reason for the introduction of the conformity relation and definition of conceptual relations was to rule out conceptual graphs which make no sense. The conformity relation insures that referents conform to the base type of a concept, while the definition of conceptual relations insures that all of the relations in a conceptual graph are linked to concepts whose base types are of the proper type. In simple terms, a conceptual graph is said to be canonical if 1) all of the referents in the graph conform to their base types; and 2) all of the conceptual relations are linked to concepts which are of a type or subtype of a type specified in the definition of the relation.

The set of all possible graphs which are canonical, is known as the canonical base. This base defines what is conceivable in the world being modeled.

2.7 Definition of Schemata

To serve as a tool for modeling the real world, conceptual graphs must be able to represent not only what is conceivable, but what is plausible, i.e., things that actually do occur in the real world. The canonical base defines all conceivable ideas; however. not all conceivable ideas are plausible ones. The canonical base rules out the idea of a green feeling (the conceptual graph [FEELING] -> (COLR) -> [GREEN]), since the relation COLR can only link PHYSOBJs to COLRs. However. the canonical basis does allow the idea of a green cow (the conceptual graph [COW] -> (COLR) -> [GREEN] is canonical, since COW is a subtype of PHYSOBJ).

To represent things that are plausible, and to represent them in a stochastic model, Sowa uses the idea of a schema. Schemata are ways of representing differing views (perspectives) of the meaning or definition of a concept. This is somewhat analogous to a dictionary definition, where the same word can have multiple definitions. Unlike type definitions, a concept can have multiple schematic definitions. For example, the concept CAT can be viewed as a MAMMAL and also as a PET. Also, unlike type definitions, schemata are neither necessary nor sufficient means for classifying an individual. Schemata are defined in much the same way as individuals and types:

```
schema for <concept name> ( <formal parameter> ) is
   <conceptual graph>
```

Just as in a type definition, the formal parameter must appear as the referent of some concept in the conceptual graph.

Using the example of a CAT being both a MAMMAL and a PET, we could write two schema definitions for cat (... means that the rest of the definition is unimportant to this discussion):

```
schema for CAT (x) is
   [MAMMAL: *x]-
   ...

schema for CAT (x) is
   [PET:*x]-
   ...
```

This essentially defines two ways of viewing a CAT, depending upon the context. The collection of all schemata

for a given concept is called its schematic cluster.

2.8 Definition of Prototypes

In the real world, in addition to viewing a concept in a number of different ways, we make generalizations about things. We often speak of a "typical" day at work, a "typical" meal, and so on. And, though everyone's idea of what is typical is different, we all do make such generalizations. Therefore, it is important for a knowledge representation scheme to do the same.

The idea of default values for describing typical individuals is a standard part of frame theory and technology. In Conceptual Graph theory, typical individuals are described through definition of prototypes. A prototype is defined in much the same way as are schemata:

```
prototype for <concept name> ( <formal parameter> ) is
   <conceptual graph>
```

Again, as with type and schema definitions, the formal parameter must occur as the referent of some concept in the conceptual graph.

As an example of a prototype, suppose we wanted to describe a typical compact car:

```
prototype for COMPACT-CAR (x) is
   [CAR: *x]-
      (CHRC) -> [WEIGHT: @ 2000 lbs]
      (CHRC) -> [LENGTH: @ 12 ft]
      (PART) -> [ENGINE]-
                   (PART) -> [CYLINDERS: {*}]-
                                (QTY) -> [NUMBER: 4],
                   (CHRC) -> [POWER: @ 75 hp],
      (PART) -> [WHEEL: {*}]-
                   (QTY) -> [NUMBER: 4].
```

This defines a typical compact car as a car with, among other things, a weight of 2000 pounds, a length of 12 feet, four wheels, and a four-cylinder engine with a power of 75 horsepower.

2.9 Putting It All Together

In this overview of conceptual graph theory, we have looked at how conceptual graphs represent some important aspects of meaning. In summary, conceptual graphs represent the following aspects of meaning:

-- Necessary and sufficient means of classification of

individuals is handled through type definitions. The types are grouped together in a type hierarchy. This is analogous to a dictionary definition, or to a database schema definition.

-- Differing views of the meaning of a concept are supported through the definition of schemata. A conceptual graph schema definition is analogous to a database subschema definition. The collection of all schemata for a given concept is called its schematic cluster.

-- Individual instances of types are represented with individual definitions. The collection of all known individuals represents what is actually in the world being modeled and corresponds to the idea of an extension in database terminology.

-- Typical individuals and default values for concepts are represented through definition of prototypes. These correspond to default values of slots in frame theory.

2.9.1 A Note About Sowa's Book and This Paper

As previously stated, Sowa's theory of Conceptual Graphs would seem to be the most complete theory of knowledge representation that we have at the present time. This paper has barely scratched the surface of all that is presented in Sowa's book. It has presented some elementary notions about Conceptual Graph theory, as well as informal syntactic definitions. It did not, however, address the question of how the graphs could be used to simulate cognitive behavior. This, as well as many other topics, are covered in great detail in Sowa's book.

CHAPTER 3 : SPECIFICATION OF TOOLS IMPLEMENTED

Referring to chapter 2 of this paper, it can be seen that the major representational components in conceptual graph theory are conceptual relations, types, schematic clusters, prototypes and individuals. As also pointed out, there is a linear notation for expressing these components. The author believes that a set of tools for storing the meaning of this linear notation would be quite useful. This section describes such a set of tools, provides a breakdown of the phases in which the tools should be implemented and specifies the parts which have been implemented by the author.

3.0 Phase 1 : Formal Specification of Syntax

The first phase in implementing a set of tools for representing meaning using the linear notation of conceptual graphs is to formally specify the syntax of the linear notation for defining conceptual relations, types, individuals, schemata and prototypes. The author has done this through the use of syntax graphs. The final product of this phase is 1) a list of tokens recognizable in conceptual graphs (see Appendix A); and 2) a set of syntax graphs which completely and formally specify the syntax of the definitions of types, schemata, individuals, prototypes and conceptual relations, in the linear notation for conceptual graphs (see Appendix B).

3.1 Phase 2 : Implementation of Scanner and Parser

The second phase of the implementation is to translate the syntax graphs into a set of scanning and parsing functions for the aforementioned definitions. The output of the scanning and parsing functions should be of the form of Lisp lists representing the definitions.

The author has implemented such a set of scanning and parsing functions. The scanner and parser were both written in Lisp (the code is included in Appendix C). The scanner returns tokens as specified in Appendix A.

The parser is similar to a recursive descent parser; parsing starts by calling the parsing function for the most complex non-terminal symbol, which proceeds to call parsing functions for less complex non-terminals. However, from the definition of a recursive descent parser given by Aho (1977) as "A parser that uses a set of recursive procedures to recognize its input with no backtracking", the parser cannot be considered as truly a recursive descent parser, since the parser does backtracking. Backtracking is necessary since the grammar written by the author for the linear form of Conceptual Graphs is not LL1 (see section 4.1 for an explanation of LL1 grammar).

The parser uses a strategy approximating that of a backward chaining rule-based system with backtracking. Calling a given parsing function "hypothesizes" that the next nonterminal symbol in the input stream is that for which the parsing function is meant to parse. If the input stream matches with the grammar being parsed, the hypothesis is taken to be true, and a representation of the meaning of that series (in the form of a Lisp list) is returned. If the parsing function doesn't find such a series of tokens, it backtracks by putting back onto the input stream the tokens taken off by the scanner during the parsing process, and returns nil to signify that the form being parsed for is not the next one on the input stream.

The individual parsing functions match one for one with the grammar represented by the set of syntax graphs appearing in Appendix B (for example, the parsing function "parse-conceptual-graph" parses the syntax specified in the syntax graph for "conceptual-graph"). The output of each of the parsing functions consists of Lisp S-expressions.

The functions are of basically two kinds: 1) those which parse by calls to other parsing functions (these are included as the first set of functions); and 2) those which parse by calls directly to the lexical scanner (the second set of functions).

Since the functions which parse by calls to other parsing functions are conceptually far removed from the actual characters of the input stream (one must get relatively far into the parse tree before it is determined whether or not the input matches the grammar being parsed), it is difficult for such a parsing function to know what characters to put back, since a record must be kept of characters taken off the input stream. Therefore. when many of the parsing functions fail to match the input stream with the syntax being parsed for, they are unable to backtrack, and so they return nil without putting characters back on the input stream. At such points, an error condition can be considered to be in effect.

On the other hand, since the functions which parse by calls directly to the lexical scanner are parsing for simple tokens (one needs not get very far into the parse tree before it is determined whether or not the input matches the grammar being parsed), it is an easy matter for them to know what characters to put back if the input stream doesn't match with the syntax they are meant to parse. Therefore, all of these functions are able to backtrack, and thus can put back onto the input stream the characters removed by the lexical scanner.

3.2 Phase 3 : Storage of Meaning of Definitions

The third phase of the implementation is to store the meaning of the Lisp lists produced by the parser into appropriate databases.

The meaning of the definitions consists of a set of components which vary according to what type of definition is being made. For example, the meaning of a type definition consists of the type name, formal parameter name, and the conceptual graph describing how the type being defined differs from its supertype. These components are listed in the comment section preceding the code in Appendix C for the parsers for each of the definitions (type, individual, relation, schema and prototype).

The meaning of a conceptual graph is represented as a list of relation1-concept1-concept2 lists, where concept1 and concept2 are concepts being linked by the conceptual relation relation1. Before storing the graphs, semantic errors should be checked for. This entails insuring that graphs being defined are canonical; see section 2.61 for a definition of what is meant by canonical. This phase probably should be combined with the second phase by modifying the parser so that it performs semantic checks and produces the appropriate definitions.

The first five functions in the parser (type, individual, relation, prototype, and schema), respectively parse type, individual, conceptual relation, prototype and schema definitions. They presently each store the results of the definitions in one of five property lists under the name of the definition. The precise form of the stored definition is documented in the comments preceding the code for each of the parsing functions, in Appendix C.

CHAPTER 4 : EVALUATION OF THE IMPLEMENTATION

The implementation of the tools for representation of knowledge using Sowa's Conceptual Graphs has gone quite well. All of the tools which the author proposed to implement have been finished on schedule. In general, the implementation has been a success. There are. however, some additions and modifications which would make the system considerably more useful. This chapter serves as a critique of the individual parts of the project and outlines at each point the additions which could be made to the system to enhance its usefulness (as well as serve as a Master's project for one or more graduate students).

4.0 The Lexical Scanner

Although a lexical scanner is theoretically an easy piece of software to implement. implementation of a scanner for the linear notation for conceptual graphs was complicated by the fact that some of the characters used in the linear form are treated specially in Franz Lisp. The author therefore spent a considerable amount of time finding how to deal with the Lisp reader's special treatment of these characters.

This problem was solved by changing the class of these problem characters so the Lisp reader treats them as ordinary characters ("vcharacters"). This obviously has side effects; in this case, several features provided by Franz Lisp no longer work (a list of the "problem" characters and the syntactic changes and side effects are documented at the beginning of the code for the scanner in Appendix C).

It would appear that there is no way to get around these side effects without radically changing the configuration and strategy of the Franz Lisp interpreter and compiler. In the author's opinion, these side effects are minor and are best simply lived with.

4.1 Formal Specification of the Grammar

Perhaps the most difficult phase of this project was to formally specify the grammar of the linear notation for Conceptual Graphs. The author did not find any such specification in existence, so this specification had to be made from examples in Sowa's book.

In the author's opinion, BNF notation for formal specification of grammar is not highly readable; therefore the specification was done through the use of syntax graphs. This definition is included as Appendix B.

As mentioned in section 3.1, the grammar which was specified was not an LL1 grammar. i.e. a grammar such that for each production, reading of one token is sufficient to

determine which production should be examined next. Although LL1 grammars are more efficient to parse (they require no backtracking) and produce parsers for which error detection and recovery is easier to implement, the author did not structure the grammar as LL1.

LL1 grammars tend to be structured with efficiency of implementation as a major goal, with ease of understanding as a secondary goal. Although the author did some such structuring to make the resulting parser easier to implement, the primary consideration was ease of understanding of the grammar. The final form of the grammar is believed to be sufficiently readable to serve as a tutorial on the syntax of the linear form of conceptual graphs.

In retrospect, the author believes that structuring the grammar as LL1 would result in a grammar which would be as readable as that which was produced. However, although the process of the specification of the grammar and the implementation of the resulting parser resulted in a grammar and parser. neither of which are ideal, it did have interesting implications toward software design methodologies.

Since the grammar was constructed from somewhat vague examples, production of the grammar and the resulting parser were carried out concurrently, in an incremental, bottom-up fashion. As parts of the grammar were specified, they were tested by writing and running the corresponding parts of the parser. This seemed to make design and implementation of the parser a more manageable and predictable task. The author believes that specification of the entire grammar before beginning work on the parser would have been a much more difficult and uncertain task, as the validity of the parts of the grammar would have been more questionable without their being tested with the corresponding parts of the parser.

The result of the specification of the grammar and the implementation of the resulting parser is a rapidly prototyped system. It took a total of seven months to go from a vague understanding of conceptual graphs to a very good understanding of conceptual graphs and a working system. The author makes no claims about the robustness of the system; robustness is not considered to be an important factor in such a prototype system as this. However, the system does serve as a valuable starting point for experimentation with implementation of ideas from Conceptual Graph theory.

4.2 The Parser

Since the grammar from phase two of the implementation was entirely recursive, it was a straightforward process to implement a modified recursive descent parser (modified by addition of backtracking, since the grammar being parsed is not LL1) to parse the grammar. The code for recursive descent

parsers is normally quite easily understood, as is the case with code for this parser. It would also be a straightforward task to make additions to the language, such as addition of features to represent procedural knowledge, as Hartley (1985) proposes to do for expert systems.

There is presently no syntactic error detection nor recovery provided for the parser. In general, when a given parsing function is called, if it cannot find any of the non-terminal symbols that it expects, it simply returns nil. The author believes that this is sufficient for such a prototype system. Error detection and recovery could be added at a later time.

Although the author did not propose to implement a mechanism for storing the results produced by the parser. the parser does currently store the Lisp lists produced on property lists. This is considered adequate for current purposes; however. the author believes that the frame structure provided by PEARL is a better choice as a storage mechanism (see section 4.3 for a more thorough explanation; for a complete treatment of the PEARL AI language, see the reference by Deering (1982)).

The parser also currently has no semantic error detection nor recovery. This allows non-canonical graphs to be stored in the databases (see section 2.61 for an explanation of the term 'canonical'). This should not be a difficult task, but is a crucial one to insure integrity of the canonical base.

4.3 The Environment

Since the major use for conceptual graphs is artificial intelligence (AI) research, it made sense to embed conceptual graphs in the preferred language of AI researchers, which is Lisp. The superior environment provided by Lisp, along with its easily extensible nature, facilitates experimentation, thus allowing the most flexible usage of the tools. Therefore, the Lisp environment was the logical choice.

Furthermore. there is a frame-based extension of Franz Lisp, called PEARL (Package for Efficient Access to Representations in Lisp) available on ksuvaxl at KSU. This system provides access to frame structures, while still in the Lisp environment. Implementation of the tools specified in section 3.0 in the PEARL environment allows access to the frame structure provided by PEARL. The author therefore implemented these tools in PEARL-enhanced Franz Lisp in the PEARL environment. However. the system presently does not make use of the frame structure provided by PEARL; lists created by the parser are currently stored on property lists.

4.4 Specific Accomplishments

This section serves as a summary of the specific accomplishments made by the author during the design and implementation of the system for analysis and storage of the meaning of conceptual graph definitions.

The first accomplishment was specification of a list of tokens of which the linear notation for Sowa's conceptual graphs are made. This specification is included in Appendix A and is also the specification of what tokens are returned by the lexical scanner.

The second accomplishment was the formal specification of the syntax of conceptual graphs. As pointed out earlier in this chapter, this was done through the use of syntax graphs, which are included in Appendix B.

The third accomplishment was the implementation of a set of scanning and parsing functions which check the syntax of type, individual, conceptual relation, prototype and schema definitions, and stores the meaning of these definitions. Code for these functions is included in Appendix C, and examples of the use of the functions is included in Appendix D.

4.5 Future Work

As mentioned earlier. the system designed and implemented by the author is a prototype system and as such is only a starting point for work on implementation of a truly robust system. This section proposes additions and modifications to the system which the author believes would increase its usefulness.

As pointed out in section 4.0, several characters which are a part of conceptual graph notation are treated specially by the Franz Lisp reader. The problem was solved by changing the syntax class of the problem characters so that the characters are no longer treated specially. However, this also eliminated some useful functions built into Franz Lisp. The author could see no way of eliminating this problem without radical modification of the Franz Lisp system itself. Solution of this problem might be worthy of some effort, although this seems doubtful.

As was mentioned in section 4.1, the grammar in Appendix B is not LL1, and therefore. it is not easy to write error recovery routines for the parser. It would be worthwhile, and not a very difficult task to restructure the grammar so that it is LL1. A thorough treatment of LL1 grammar is given in Aho (1977), which should aid in this restructuring.

Once the grammar is restructured to LL1, it will be necessary to modify the parser to reflect these changes. At the same time, it will be a fairly straightforward task to add syntactic error detection and recovery to the parser.

As also pointed out, the parser does no semantic checks on the definitions made. This would entail, among other things, checking definitions to assure canonicality of the graphs and checking to assure that the definitions being made haven't already been made.

Sowa (1984) provides a limited discussion of the distinction between declarative and procedural knowledge, and proposes a notation for representation of procedural knowledge. He proposes a notation of dataflow graphs, which contain actors which link conceptual graphs. Hartley (1985) proposes adaptation of this notation for representation of procedural knowledge in expert systems. The author believes that addition of this notation to the grammar and parser would be a large step forward in the effort to develop tools for representation of procedural knowledge.

Once this addition of procedural knowledge-handling capabilities to the scanner and parser are complete, the frame-based capabilities of the PEARL AI language would become quite important. as the active value (demon) handling functions provided by PEARL would be ideally suited to carry out the state transformations which occur in the data flow graphs. Therefore. it would be advantageous to switch from storing the meaning of the definitions on property lists, to storing them in PEARL frames.

THIS BOOK
CONTAINS
NUMEROUS PAGES
WITH THE ORIGINAL
PRINTING BEING
SKEWED
DIFFERENTLY FROM
THE TOP OF THE
PAGE TO THE
BOTTOM.

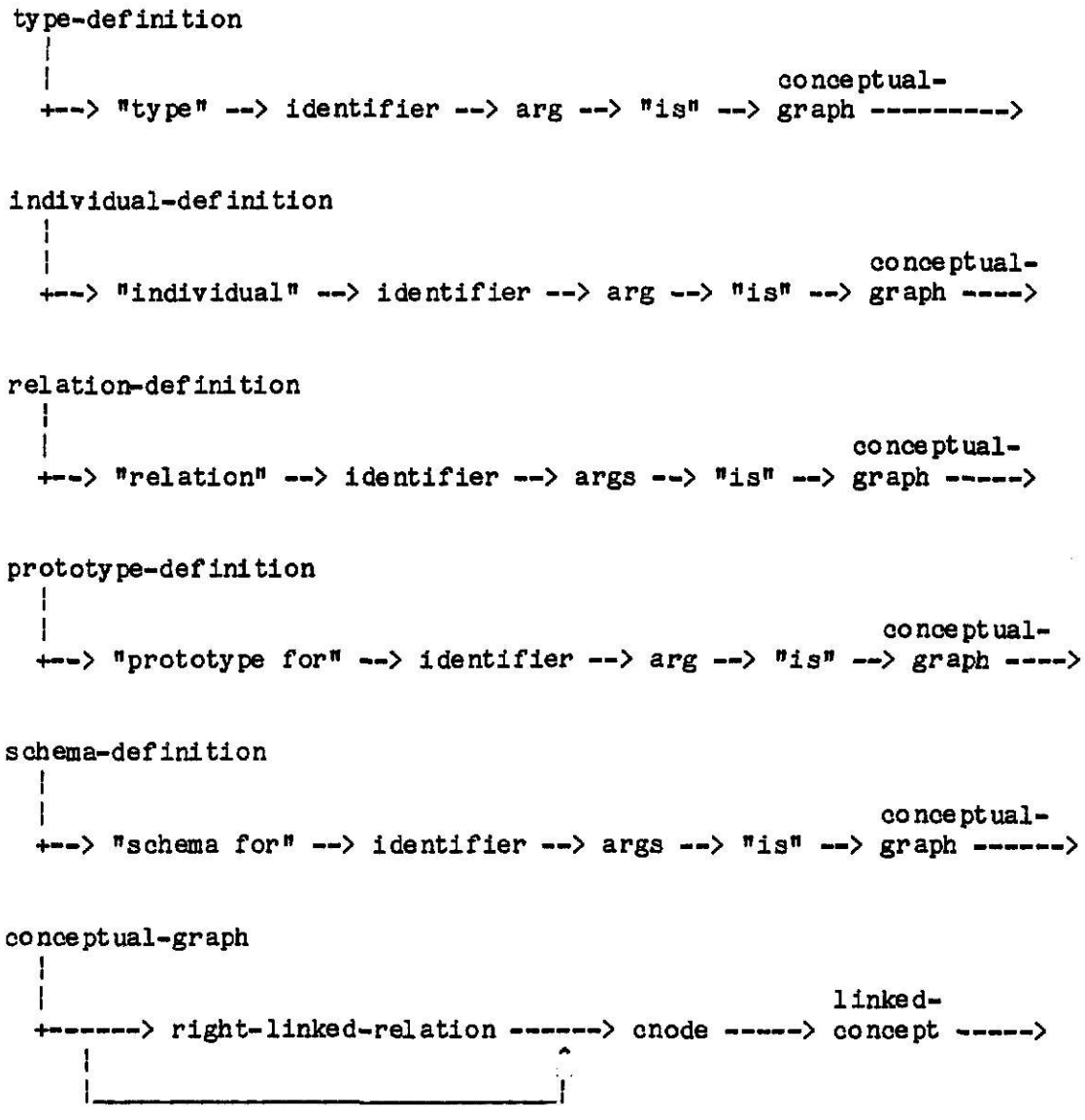
THIS IS AS RECEIVED
FROM THE
CUSTOMER.

Appendix A : Tokens recognized by the scanner

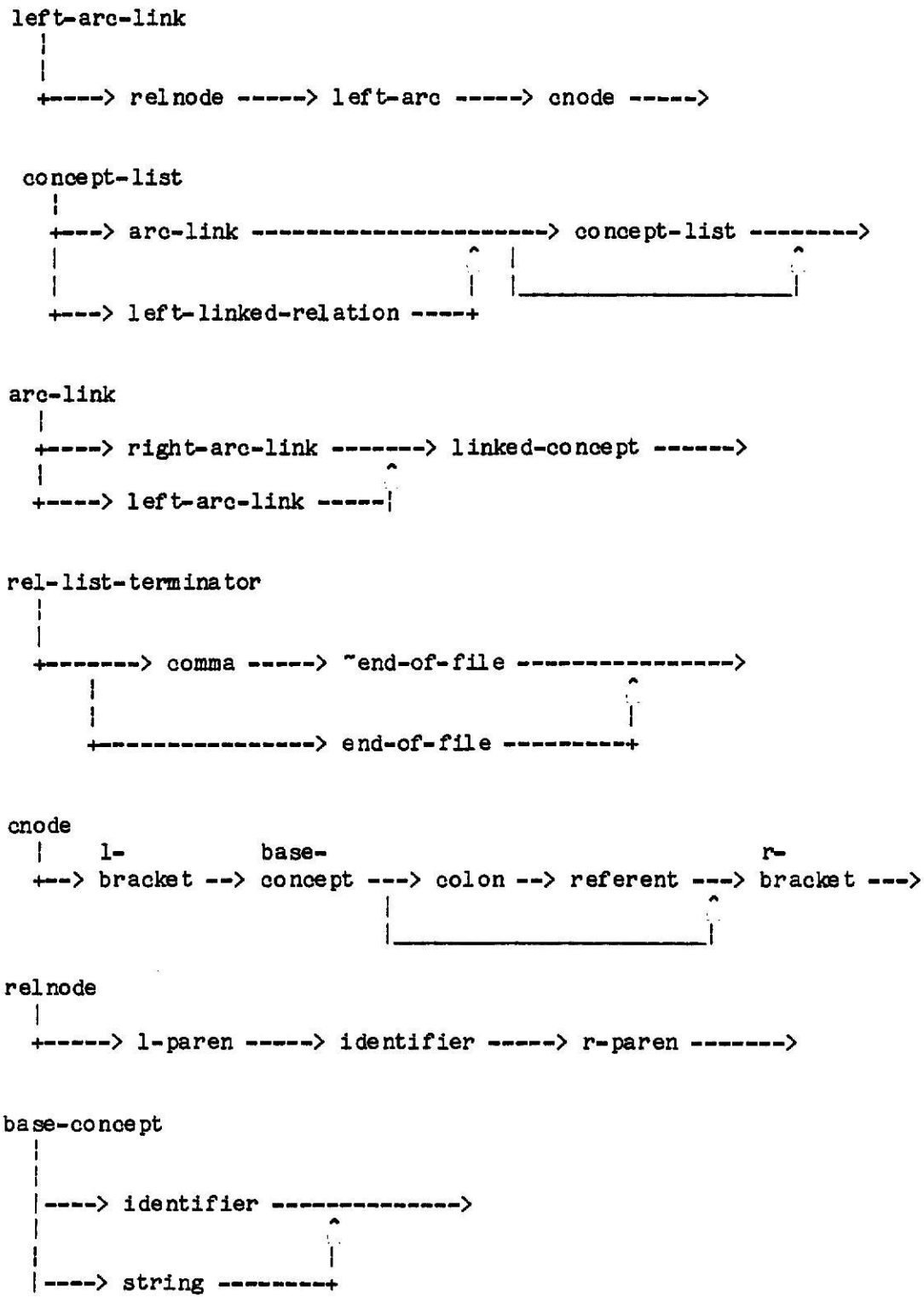
[	left-bracket
]	right-bracket
(	left-paren
)	right-paren
{	left-brace
}	right-brace
:	colon
,	comma
.	period
#	sharp-sign
*	asterisk
@	at sign
¦	vertical-bar
>	lt sign
"	double-quote
-	bi-dir-arc
->	right-arc
<-	left-arc
	identifier
	integer

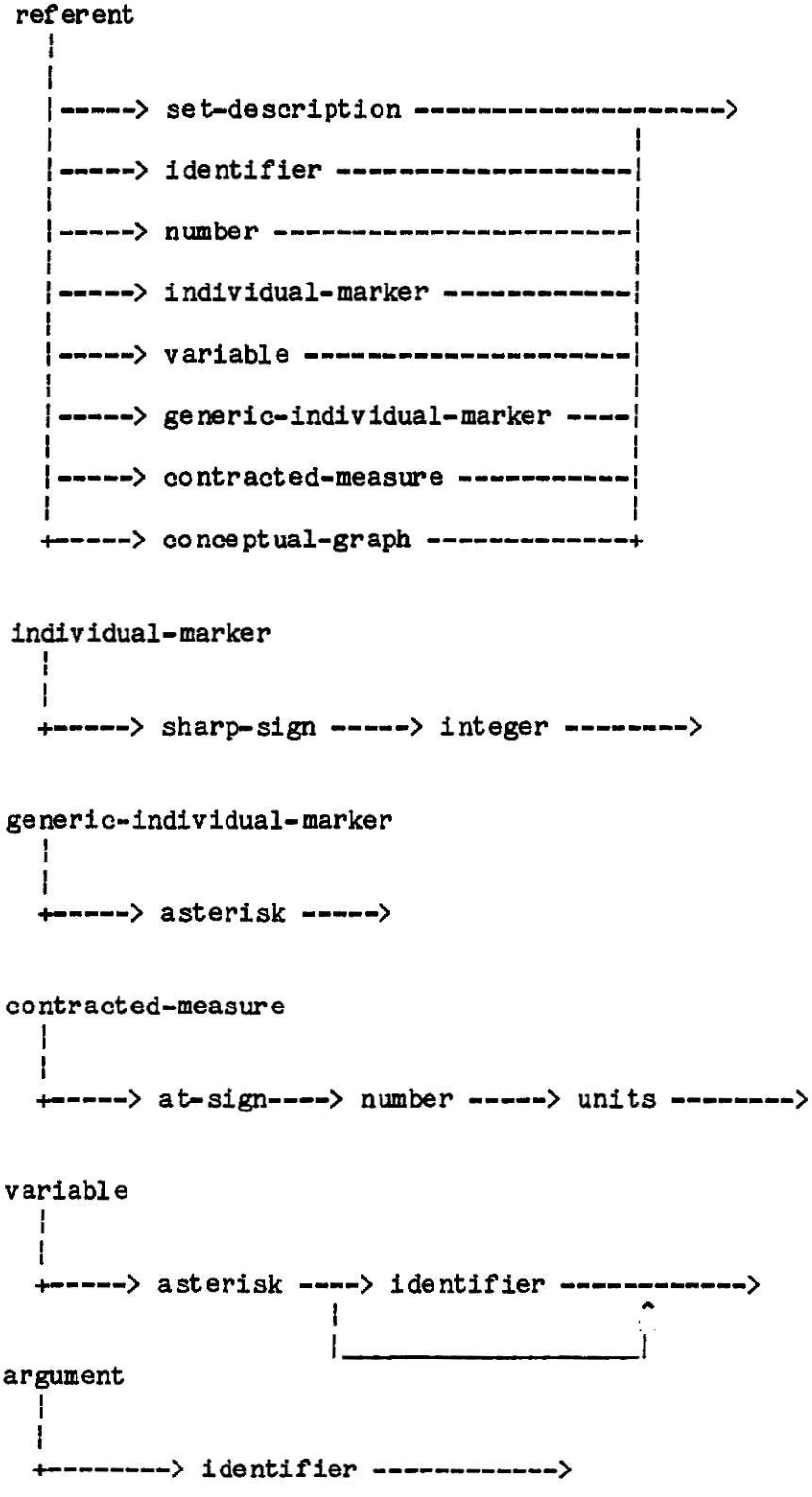
Appendix B : Syntax Graphs for Conceptual Graphs

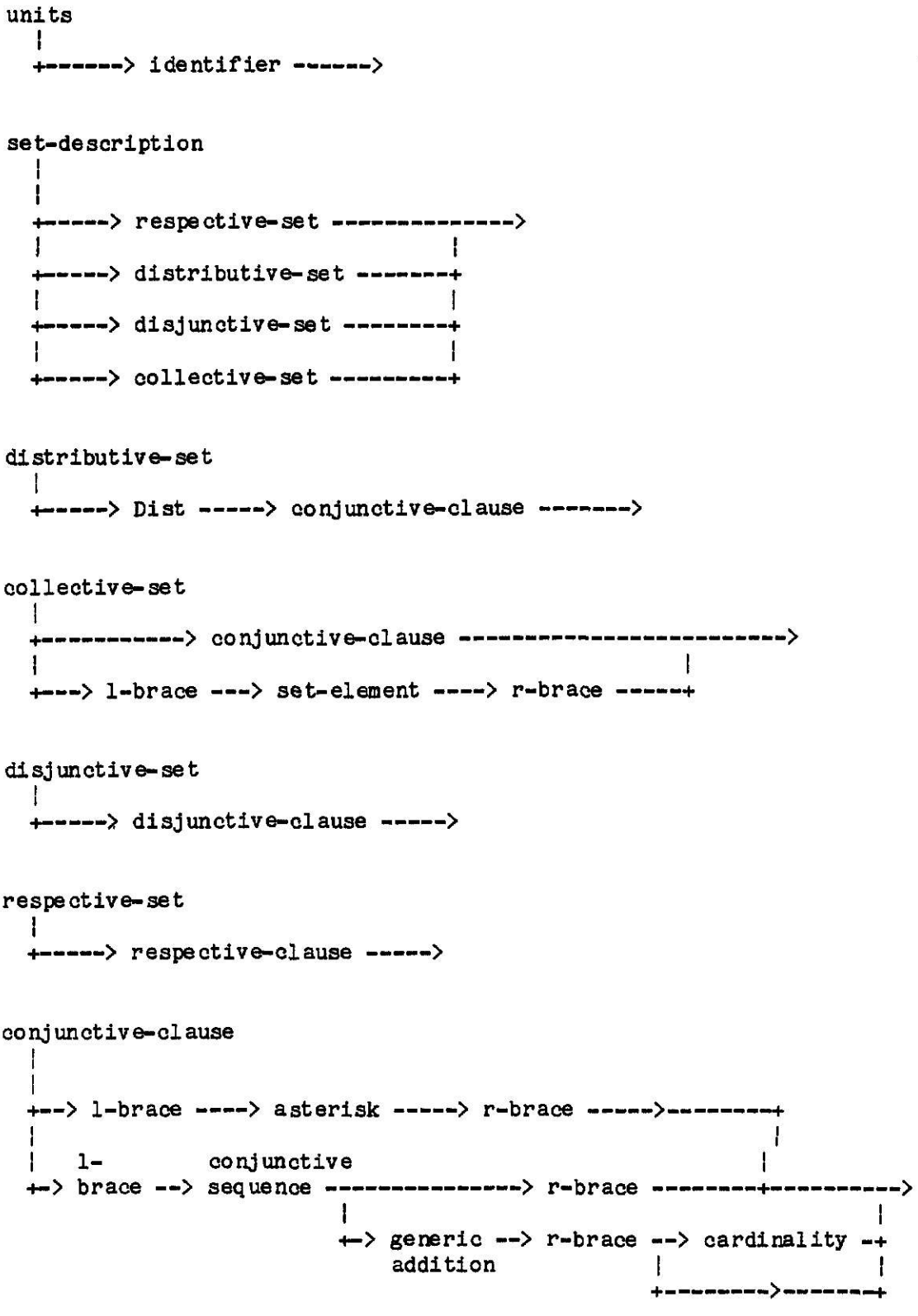
TABLE OF CONTENTS FOR APPENDIX B

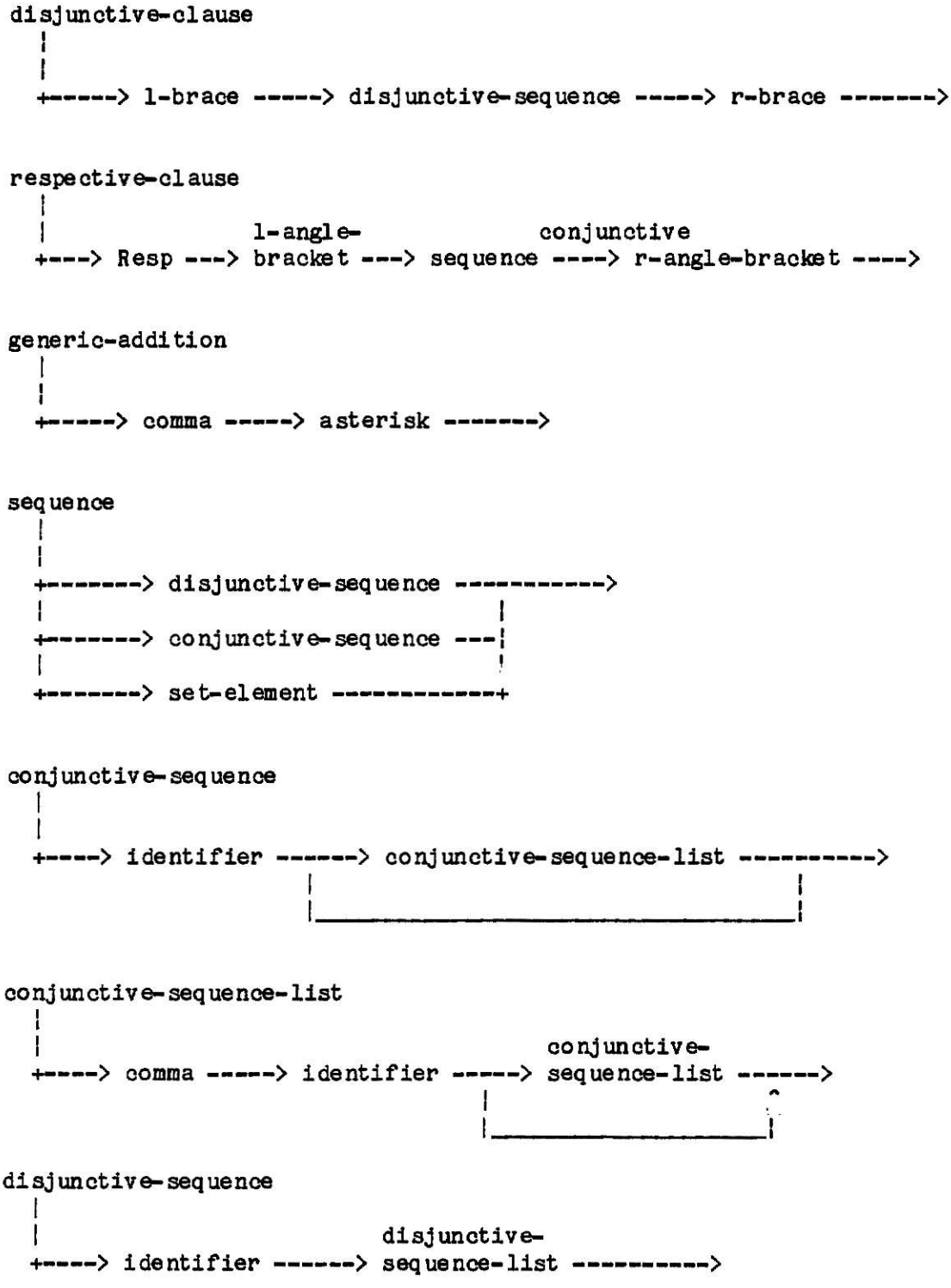

type-definition
"type"
identifier
arg
"is"
conceptual-graph
individual-definition
"individual"
identifier
arg
"is"
conceptual-graph
relation-definition
"relation"
identifier
args
"is"
conceptual-graph
prototype-definition
"prototype for"
identifier
arg
"is"
conceptual-graph
schema-definition
"schema for"
identifier
args
"is"
conceptual-graph
conceptual-graph
right-linked-relation
cnode
linked-concept

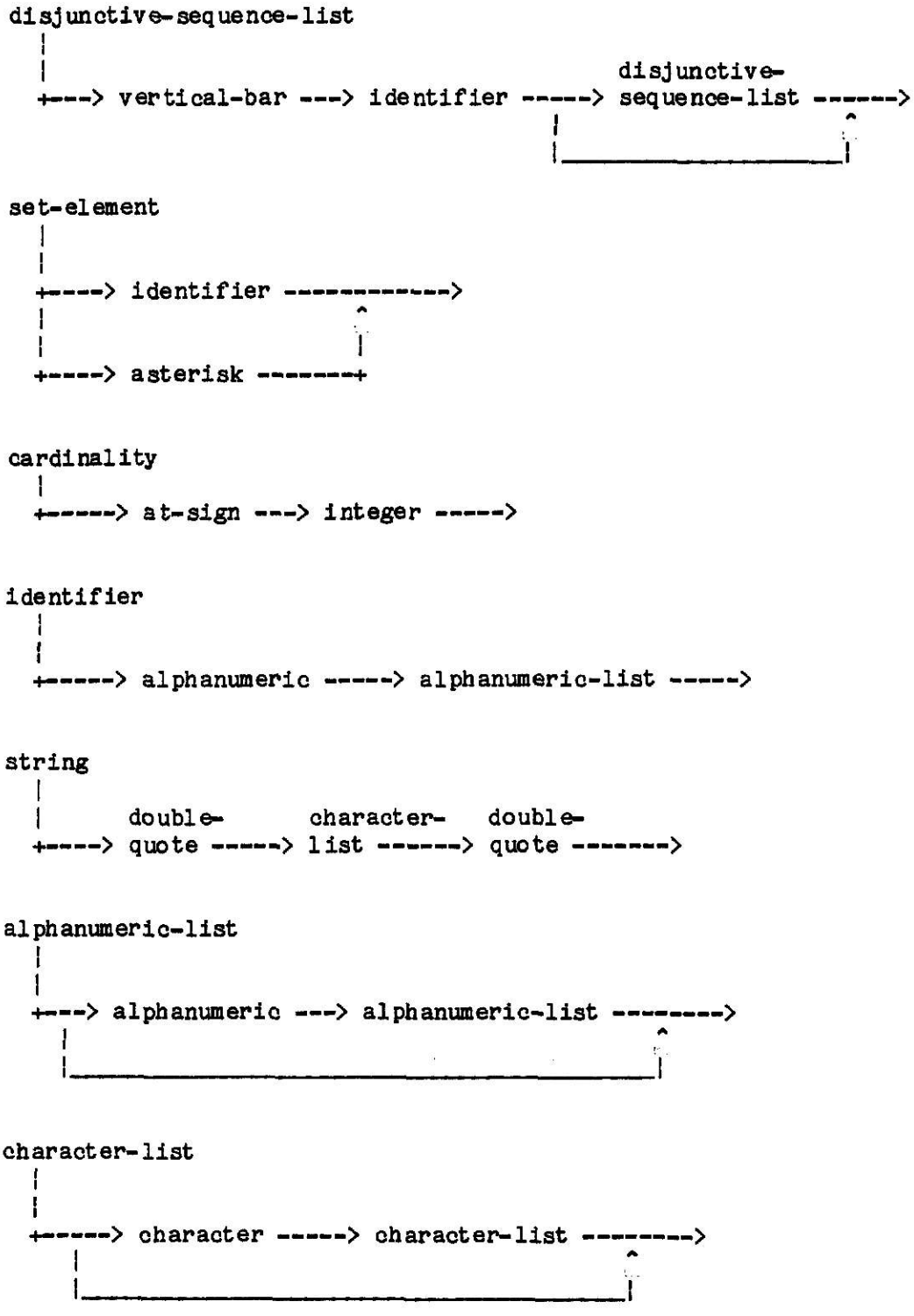

linked-concept
right-linked-concept
left-linked-concept
bi-dir-concept-list
right-linked-concept
right-arc
right-arc-link
linked-concept
left-linked-concept
left-arc
left-arc-link
linked-concept
left-linked-relation
left-arc
relnode
right-linked-relation
relnode
right-arc
bi-dir-concept-list
bi-dir-arc
concept-list
rel-list-terminator
right-arc-link
relnode
right-arc
cnode

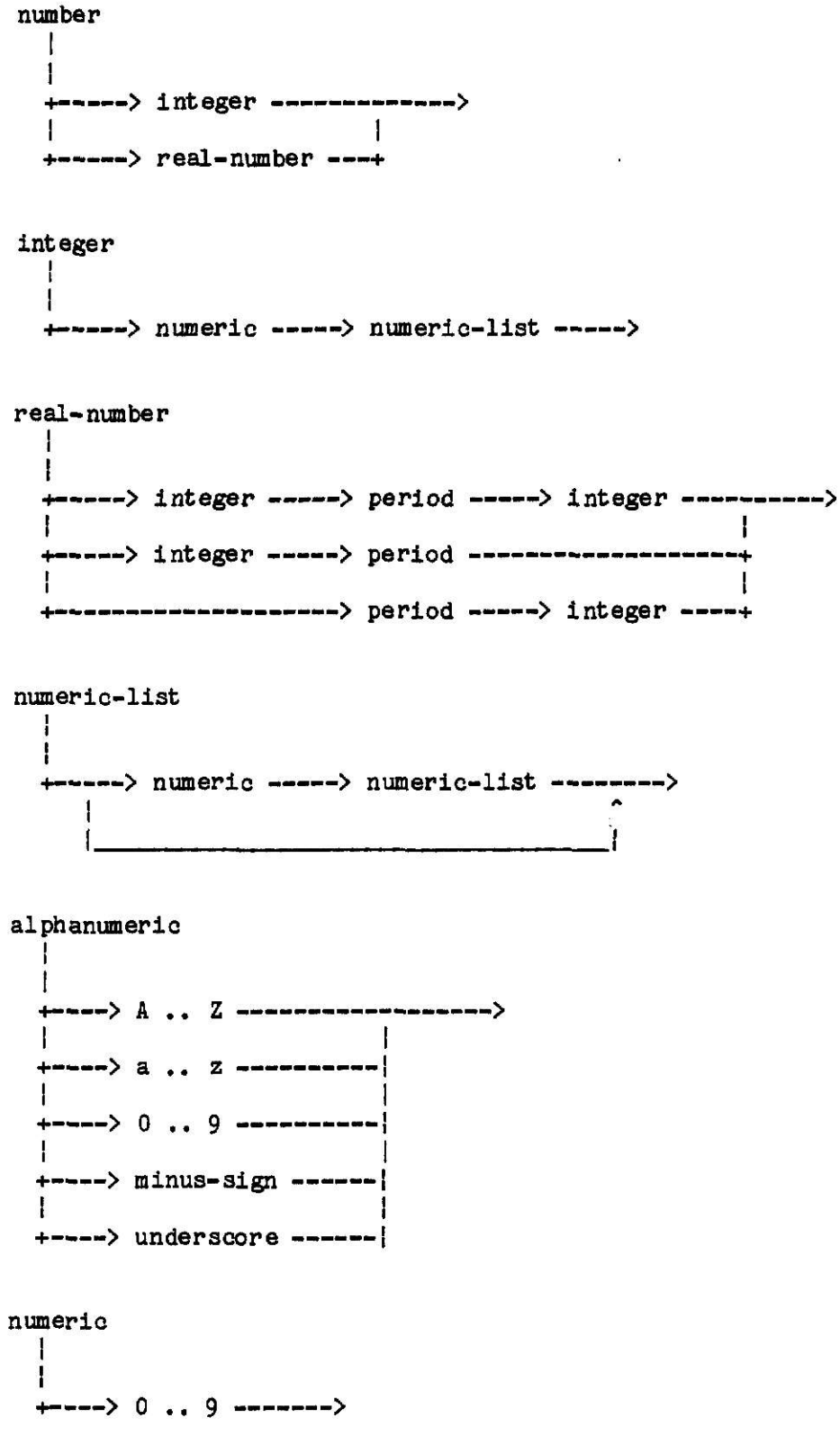

left-arc-link
relnode
left-arc
cnode
concept-list
arc-link
concept-list
left-linked-relation
arc-link
right-arc-link
linked-concept
left-arc-link
rel-list-terminator
comma
~end-of-file
end-of-file
cnode
l-bracket
base-concept
colon
referent
r-bracket
relnode
l-paren
identifier
r-paren
base-concept
identifier
string

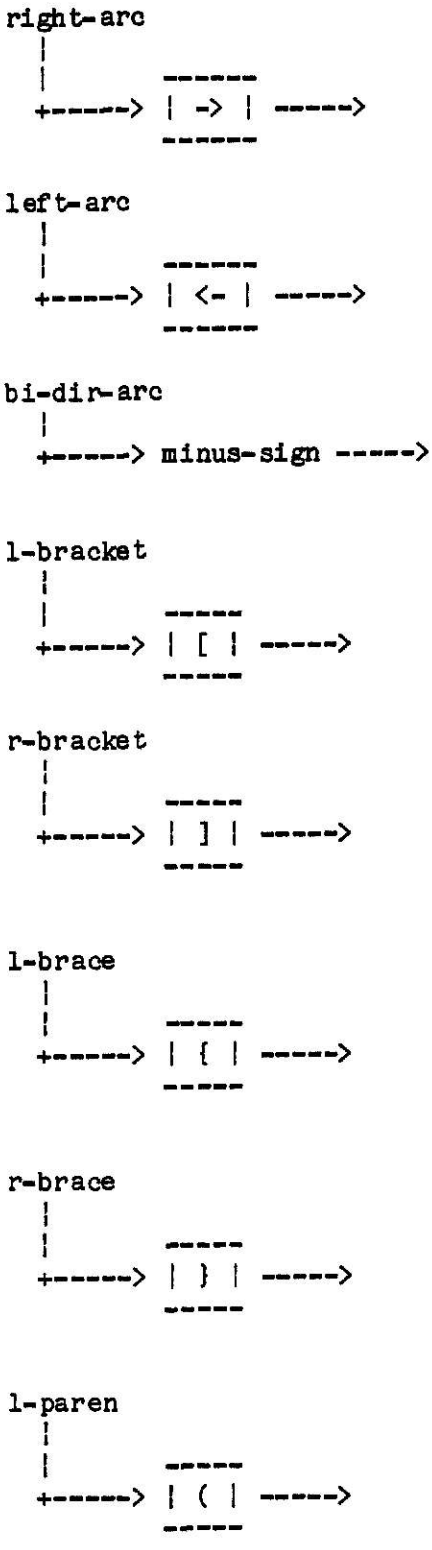

referent
set-description
identifier
number
individual-marker
variable
generic-individual-marker
contracted-measure
conceptual-graph
individual-marker
sharp-sign
integer
generic-individual-marker
asterisk
contracted-measure
at-sign
number
units
variable
asterisk
identifier
argument
identifier

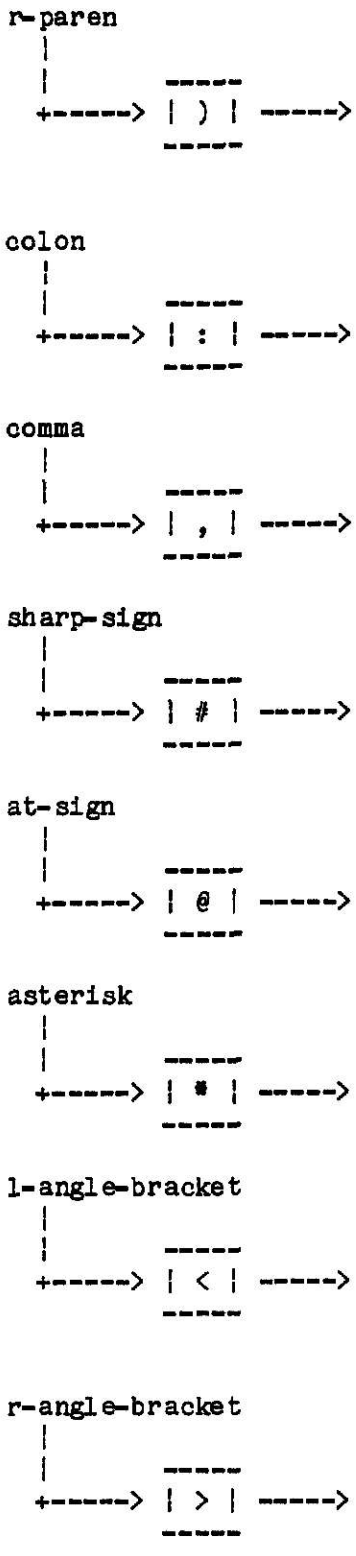

units
identifier
set-description
respective-set
distributive-set
disjunctive-set
collective-set
distributive-set
Dist
conjunctive-clause
collective-set
conjunctive-clause
l-brace
set-element
r-brace
disjunctive-set
disjunctive-clause
respective-set
respective-clause
conjunctive-clause
l-brace
asterisk
r-brace
l-brace
conjunctive sequence
r-brace
generic addition
r-brace
cardinality

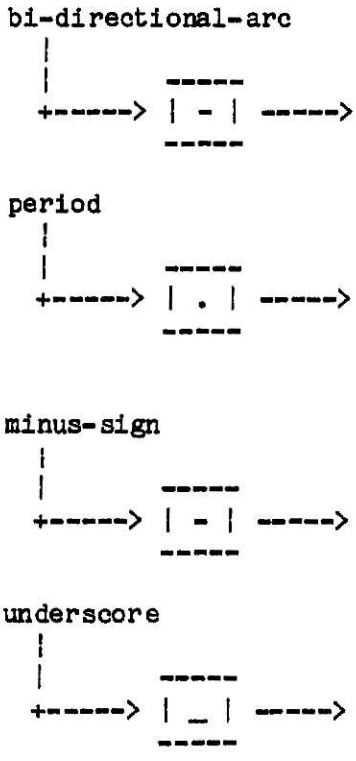

disjunctive-clause
l-brace
disjunctive-sequence
r-brace
respective-clause
Resp
l-angle-bracket
sequence
conjunctive
r-angle-bracket
generic-addition
comma
asterisk
sequence
disjunctive-sequence
conjunctive-sequence
set-element
conjunctive-sequence
identifier
conjunctive-sequence-list
conjunctive-sequence-list
comma
identifier
conjunctive-sequence-list
disjunctive-sequence
identifier
disjunctive-sequence-list


disjunctive-sequence-list
vertical-bar
identifier
disjunctive-sequence-list
set-element
identifier
asterisk
cardinality
at-sign
integer
identifier
alphanumeric
alphanumeric-list
string
double-quote
character-list
double-quote
alphanumeric-list
alphanumeric
alphanumeric-list
character-list
character
character-list


number
integer
real-number
integer
numeric
numeric-list
real-number
integer
period
integer
integer
period
period
integer
numeric-list
numeric
numeric-list
alphanumeric
A .. Z
a .. z
0 .. 9
minus-sign
underscore
numeric
0 .. 9


right-arc
| -> |
left-arc
| <- |
bi-dir-arc
minus-sign
l-bracket
[
r-bracket
]
l-brace
{
r-brace
}
l-paren
(

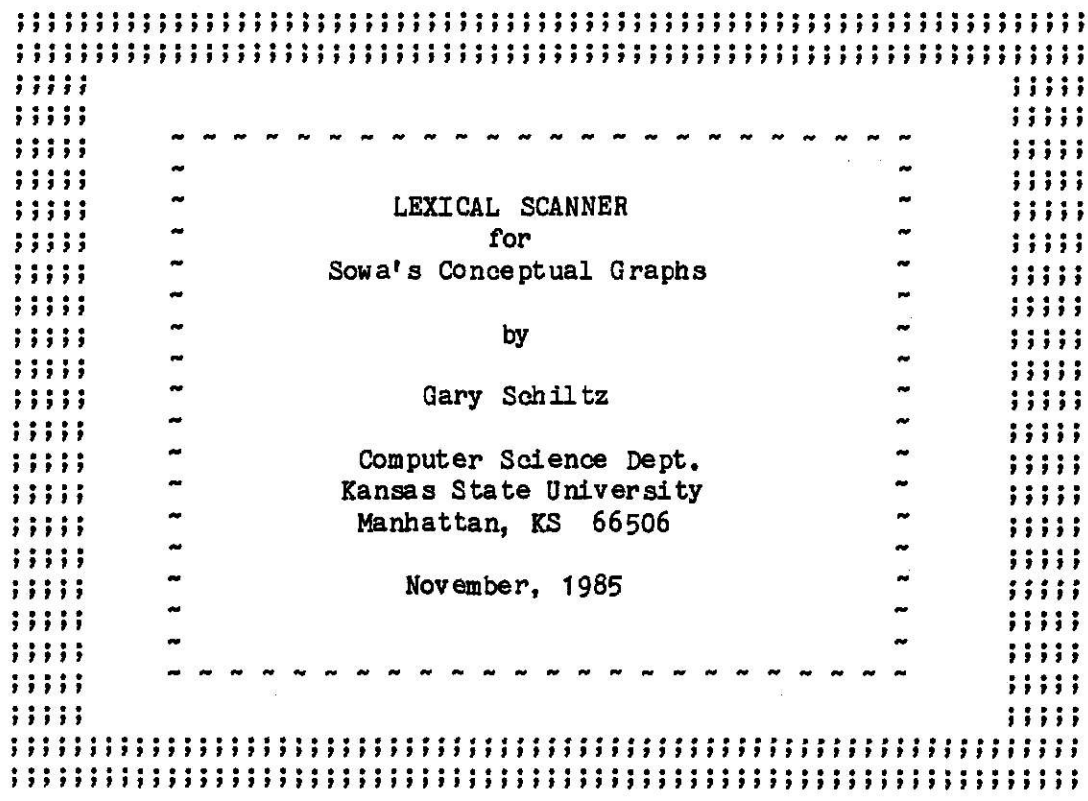

r-paren
|) |
colon
| : |
comma
| , |
sharp-sign
| # |
at-sign
| @ |
asterisk
| * |
l-angle-bracket
| < |
r-angle-bracket
| > |


bi-directional-arc
| - |
period
| . |
minus-sign
| - |
underscore
| _ |

Appendix C : Source Code for Scanner and Parser

This appendix contains the source code for the scanner and parser for analysis of the syntax of the linear form of Conceptual Graphs.

TABLE OF CONTENTS FOR APPENDIX C


LEXICAL SCANNER
for
Sowa's Conceptual Graphs
by
Gary Schiltz
Computer Science Dept.
Kansas State University
Manhattan, KS 66506
November, 1985

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;;;                                                         ;;;;;
;;;;;  The following set of functions makes up the lexical    ;;;;;
;;;;;  scanner for tokenizing the linear form of Sowa's       ;;;;;
;;;;;  conceptual graph notation.                             ;;;;;
;;;;;                                                         ;;;;;
;;;;;  The types of tokens returned by the scanner are as     ;;;;;
;;;;;  follows:                                               ;;;;;
;;;;;                                                         ;;;;;
;;;;;          Tokens recognized by the scanner               ;;;;;
;;;;;          --------------------------------               ;;;;;
;;;;;                                                         ;;;;;
;;;;;                  [    left-bracket                      ;;;;;
;;;;;                  ]    right-bracket                     ;;;;;
;;;;;                  (    left-paren                        ;;;;;
;;;;;                  )    right-paren                       ;;;;;
;;;;;                  {    left-brace                        ;;;;;
;;;;;                  }    right-brace                       ;;;;;
;;;;;                  :    colon                             ;;;;;
;;;;;                  ,    comma                             ;;;;;
;;;;;                  .    period                            ;;;;;
;;;;;                  #    sharp-sign                        ;;;;;
;;;;;                  *    asterisk                          ;;;;;
;;;;;                  @    at sign                           ;;;;;
;;;;;                  |    vertical-bar                      ;;;;;
;;;;;                  >    lt sign                           ;;;;;
;;;;;                  "    double-quote                      ;;;;;
;;;;;                  -    bi-dir-arc                        ;;;;;
;;;;;                  ->   right-arc                         ;;;;;
;;;;;                  <-   left-arc                          ;;;;;
;;;;;                       identifier                        ;;;;;
;;;;;                       integer                           ;;;;;
;;;;;                                                         ;;;;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;


;
; A summary of syntactic changes necessary for the lisp reader to be able
; to read the linear form of a syntactically valid conceptual graph:
;
;                        Side effect
;                                   -----------
(setsyntax '\[ 'vcharacter) ;eliminates left superbracket
(setsyntax '\] 'vcharacter) ;eliminates right superbracket
(setsyntax '\, 'vcharacter) ;eliminates backquote function
(setsyntax '\# 'vcharacter) ;eliminates splicing macros
(setsyntax '\| 'vcharacter)

(declare (special INPUT))
```

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;                                                                ;;
;;                                                                ;;
;;                   *******  next-token  *******                 ;;
;;                                                                ;;
;;   INPUT(S) : There are no input variables accessed directly by ;;
;;              this function; input comes through calls to the   ;;
;;              function next-char.                               ;;
;;                                                                ;;
;;   RETURNS  : If there are characters left in the input stream, ;;
;;              returns a list consisting of two parts:  1) the   ;;
;;              type of token which was found; and  2) if the     ;;
;;              type of token which was found is an integer,      ;;
;;              identifier or string, its value is returned       ;;
;;              (otherwise, the list has no second part).         ;;
;;                                                                ;;
;;              If there are no characters remaining in the input ;;
;;              stream, nil is returned.                          ;;
;;                                                                ;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(defun next-token ()
   (let ((current-char (next-char)))
        (caseq current-char
        (\    (next-token))  ; ignore space character
        (\      (next-token))  ; ignore tab character
        (\\   (next-token))  ; ignore escape character
        ('\"  (scan-string))
           (\-   (scan-right-or-bidirectional-arc))
           (\<   (scan-left-arc-or-respective-set))
        ((A B C D E F G H I J K L M N O P Q R S T U V W X Y Z
          a b c d e f g h i j k l m n o p q r s t u v w x y z)
            (scan-identifier current-char))
        ((\0 \1 \2 \3 \4 \5 \6 \7 \8 \9)
            (scan-integer current-char))
        (t (list current-char)))))


(defun unget-token (token)
   (cond ((car token)
            (cond ((null (cdr token))
                   (setq INPUT (append
                         (explode (car token))
                         INPUT)))
               ((setq INPUT (append
                      (explode (cadr token))
                      INPUT)))))) nil)
```

```
(defun scan-identifier (ch)
   (list 'identifier (implode (append (list ch) (scan-id-list)))))


(defun scan-id-list ()
   (let ((ch))
     (cond ((not (alphanumeric (setq ch (next-char))))
            (cond ((not (null ch)) (unget-char ch))))
            ((append (list ch) (scan-id-list))))))


(defun scan-integer (ch)
   (list 'integer (implode (append (list ch) (scan-integer-list)))))


(defun scan-integer-list ()
   (let ((ch))
     (cond ((not (numeric (setq ch (next-char))))
            (cond ((not (null ch)) (unget-char ch))))
            ((append (list ch) (scan-integer-list))))))


(defun alphanumeric (ch)
  (memq ch '(A B C D E F G H I J K L M N O P Q R S T U V W X Y Z
          a b c d e f g h i j k l m n o p q r s t u v w x y z
          \0 \1 \2 \3 \4 \5 \6 \7 \8 \9 \0 \- \_ \@ \# \$ \% \^
          \& \* \+ \/ \? \. \!)))


(defun numeric (ch)
   (cond ((memq ch '(\0 \1 \2 \3 \4 \5 \6 \7 \8 \9 \0)) t)))


(defun scan-string ()
   (let ((char-list))
     (cond ((null INPUT) nil)
              ((null (setq char-list (scan-character-list))) nil)
           ((not (equal (next-char) '\")) nil)
           ((list 'string (implode char-list))))))

(defun scan-character-list ()
   (let ((ch))
     (cond ((equal (setq ch (next-char)) '\") (unget-char ch))
           ((append (list ch) (scan-character-list))))))
```

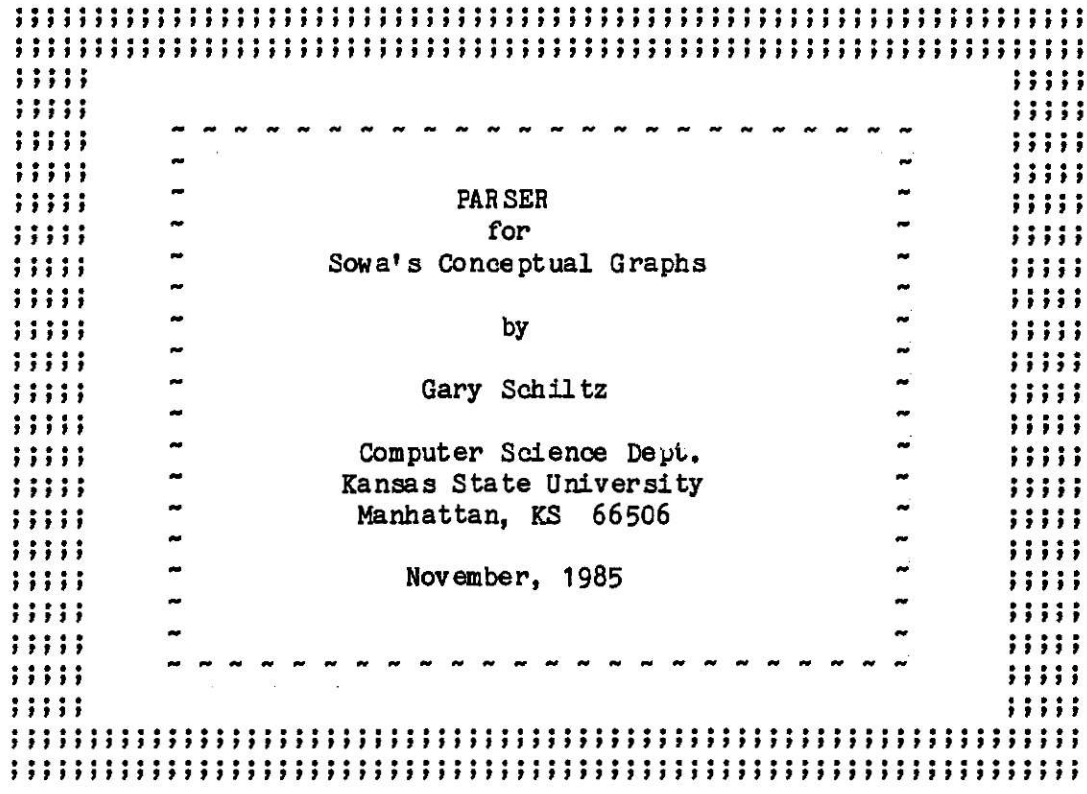
```
(defun scan-left-arc-or-respective-set ()
   (let ((current-char (next-char)))
        (caseq current-char
           ( '\-   '(<-))     ; left arc
           ( t
            (unget-char current-char)
              '(<)))))    ; left respective set delimiter

(defun scan-right-or-bidirectional-arc ()
   (let ((current-char (next-char)))
     (cond ((eq current-char '>) '(->))   ; right arc
           (t
           (unget-char current-char)
           '(-)))))                ; bidirectional arc

(defun next-char ()
   (let ((next (car INPUT)))
     (setq INPUT (cdr INPUT))
       next))

(defun unget-char (char)
   (setq INPUT (append (list char) INPUT))
   nil)

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;;                                                                          ;;;;
;;;;         ******  END OF LEXICAL SCANNER  ******                          ;;;;
;;;;                                                                          ;;;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
```


PARSER
for
Sowa's Conceptual Graphs
by
Gary Schiltz
Computer Science Dept.
Kansas State University
Manhattan, KS 66506
November, 1985

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;;;                                                           ;;;;;
;;;;;  The following functions parse the linear notation for    ;;;;;
;;;;;  conceptual graphs, which appears in John Sowa's book     ;;;;;
;;;;;  "Conceptual Structures: Information Processing in Mind   ;;;;;
;;;;;  and Machine".  The individual parsing functions match    ;;;;;
;;;;;  one for one with the grammar represented by the set of   ;;;;;
;;;;;  syntax graphs appearing in the Master's report of the    ;;;;;
;;;;;  author.  The output of each of the parsing functions     ;;;;;
;;;;;  consists of lisp S-expressions, the form of which is     ;;;;;
;;;;;  described in the explanation preceding each function.    ;;;;;
;;;;;                                                           ;;;;;
;;;;;  The functions are of basically two kinds:  1) those      ;;;;;
;;;;;  which parse by calls to other parsing functions          ;;;;;
;;;;;  (these are included as the first set of functions);      ;;;;;
;;;;;  and  2) those which parse by calls directly to the       ;;;;;
;;;;;  lexical scanner (the second set of functions).           ;;;;;
;;;;;                                                           ;;;;;
;;;;;  The parser is basically of a recursive descent type;     ;;;;;
;;;;;  parsing starts with the most complex non-terminal        ;;;;;
;;;;;  symbol, and proceeds by calling parsers for less         ;;;;;
;;;;;  complex non-terminals.                                   ;;;;;
;;;;;                                                           ;;;;;
;;;;;  There is presently no error detection nor recovery       ;;;;;
;;;;;  provided.  In general, when a given parsing function     ;;;;;
;;;;;  is called, if it cannot find any of the non-terminal     ;;;;;
;;;;;  symbols that it expects, it simply returns nil.  The     ;;;;;
;;;;;  author believes that this is sufficient for such a       ;;;;;
;;;;;  prototype system.  Error detection and recovery could    ;;;;;
;;;;;  quite easily be added at a later time.                   ;;;;;
;;;;;                                                           ;;;;;
;;;;;  The first five functions (type, individual, relation,    ;;;;;
;;;;;  prototype, and schema), parse type, individual,          ;;;;;
;;;;;  relation, prototype and schema definitions.  They        ;;;;;
;;;;;  presently store the results of the definitions in        ;;;;;
;;;;;  property lists under the name of the definition.         ;;;;;
;;;;;                                                           ;;;;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
```

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;                                                                   ;;
;;                          ***** type *****                         ;;
;;                                                                   ;;
;;   This function parses a syntactically legal type definition     ;;
;;   and stores the results in a property list.                      ;;
;;                                                                   ;;
;;   The syntax of a type definition is:                             ;;
;;                                                                   ;;
;;          ( type <type-name> ( <argument> ) is                     ;;
;;              <conceptual-graph> )                                 ;;
;;                                                                   ;;
;;   If the definition is syntactically legal, a list containing    ;;
;;   the argument and the conceptual graph is stored on the          ;;
;;   property list TYPE, under the value of type-name.               ;;
;;   Otherwise, nil is returned.                                     ;;
;;                                                                   ;;
;;                                                                   ;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(defun type fexpr (input)
   (let ((type-name) (argument) (token) (graph))
     (setq INPUT (reverse (cdr (reverse (cdr (explode input))))))

     (cond ((null (setq type-name (parse-identifier)))   nil)
           ((null (setq argument (parse-arg)))           nil)
           ((null (setq token (parse-identifier)))       nil)
           ((not  (equal (cadr token) 'is))              nil)
           ((null (setq graph (parse-conceptual-graph))) nil)

           (t (putprop 'TYPE
                 (list argument graph)
                    (cadr type-name))))))
```

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;                                                                  ;;
;;                     ##### individual #####                       ;;
;;                                                                  ;;
;;   This function parses a syntactically legal individual          ;;
;;   definition and stores the results in a property list.          ;;
;;                                                                  ;;
;;   The syntax of an individual definition is:                     ;;
;;                                                                  ;;
;;          ( individual <type-name> ( <individual-name> ) is       ;;
;;              <conceptual-graph> )                                ;;
;;                                                                  ;;
;;   If the definition is syntactically legal, a list containing    ;;
;;   the type name and the conceptual graph is stored on the        ;;
;;   property list INDIVIDUAL, under the value of individual-name.;;
;;   Otherwise, nil is returned.                                    ;;
;;                                                                  ;;
;;                                                                  ;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(defun individual fexpr (input)
   (let ((type-name) (individual-name) (token) (graph))
     (setq INPUT (reverse (cdr (reverse (cdr (explode input))))))

     (cond ((null (setq type-name (parse-identifier)))    nil)
           ((null (setq individual-name (parse-arg)))     nil)
           ((null (setq token (parse-identifier)))        nil)
           ((not  (equal (cadr token) 'is))               nil)
           ((null (setq graph (parse-conceptual-graph)))  nil)

           (t (putprop 'INDIVIDUAL
                 (list type-name graph)
                    (car individual-name))))))
```

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;                                                                    ;;
;;                   ***** relation *****                             ;;
;;                                                                    ;;
;;   This function parses a syntactically legal conceptual            ;;
;;   relation definition and stores the results in a property         ;;
;;   list.                                                            ;;
;;                                                                    ;;
;;   The syntax of a conceptual relation definition is:               ;;
;;                                                                    ;;
;;      ( relation <relation-name> ( <arguments> ) is                 ;;
;;           <conceptual-graph> )                                     ;;
;;                                                                    ;;
;;   If the definition is syntactically legal, a list containing      ;;
;;   the arguments and the conceptual graph is stored on the          ;;
;;   property list RELATION, under the value of relation-name.        ;;
;;   Otherwise, nil is returned.                                      ;;
;;                                                                    ;;
;;                                                                    ;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(defun relation fexpr (input)
   (let ((relation-name) (args) (token) (graph))
     (setq INPUT (reverse (cdr (reverse (cdr (explode input))))))

     (cond ((null (setq relation-name (parse-identifier)))  nil)
           ((null (setq args (parse-args)))                 nil)
           ((null (setq token (parse-identifier)))          nil)
           ((not (equal (cadr token) 'is))                  nil)
           ((null (setq graph (parse-conceptual-graph)))    nil)

           ((putprop 'RELATION
               (list args graph)
                  (cadr relation-name))))))
```

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;                                                                  ;;
;;                    ***** prototype *****                         ;;
;;                                                                  ;;
;;   This function parses a syntactically legal prototype           ;;
;;   definition and stores the results in a property list.          ;;
;;                                                                  ;;
;;   The syntax of a prototype definition is:                       ;;
;;                                                                  ;;
;;      ( prototype for <prototype-name> ( <argument> ) is          ;;
;;           <conceptual-graph> )                                   ;;
;;                                                                  ;;
;;   If the definition is syntactically legal, a list containing    ;;
;;   the arguments and the conceptual graph is appended to the      ;;
;;   property list PROTOTYPE, under the value of prototype-name.    ;;
;;   Otherwise, nil is returned.                                    ;;
;;                                                                  ;;
;;                                                                  ;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(defun prototype fexpr (input)
   (let ((prototype-name) (arg) (token) (graph))
     (setq INPUT (reverse (cdr (reverse (cdr (explode input))))))

     (cond ((null (setq token (parse-identifier)))              nil)
           ((not (equal (cadr token) 'for))                     nil)
           ((null (setq prototype-name (parse-identifier)))     nil)
           ((null (setq arg (parse-arg)))                       nil)
           ((null (setq token (parse-identifier)))              nil)
           ((not (equal (cadr token) 'is))                      nil)
           ((null (setq graph (parse-conceptual-graph)))        nil)

           (t (putprop 'PROTOTYPE
                 (list arg graph)
                    (cadr prototype-name))))))
```

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;                                                                 ;;
;;                       ***** schema *****                        ;;
;;                                                                 ;;
;;    This function parses a syntactically legal schema            ;;
;;    definition and stores the results in a property list.        ;;
;;                                                                 ;;
;;    The syntax of a schema definition is:                        ;;
;;                                                                 ;;
;;       ( schema for <schema-name> ( <argument> ) is              ;;
;;            <conceptual-graph> )                                 ;;
;;                                                                 ;;
;;    If the definition is syntactically legal, a list containing  ;;
;;    the arguments and the conceptual graph is appended to the    ;;
;;    property list SCHEMA, under the value of schema-name.        ;;
;;    Otherwise, nil is returned.                                  ;;
;;                                                                 ;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(defun schema fexpr (input)
   (let ((schema-name) (arg) (token) (graph))
     (setq INPUT (reverse (cdr (reverse (cdr (explode input))))))
     (cond ((null (setq token (parse-identifier))) nil)
           ((not (equal (cadr token) 'for)) nil)
           ((null (setq schema-name (parse-identifier))) nil)
           ((null (setq arg (parse-arg))) nil)
           ((null (setq token (parse-identifier))) nil)
           ((not (equal (cadr token) 'is)) nil)
           ((null (setq graph (parse-conceptual-graph))) nil)
           ((putprop 'SCHEMA
               (list arg graph)
                  (cadr schema-name))))))
```

```
(defun parse-arg ()
   (let ((id-token))
     (cond ((null (parse-l-paren)) nil)
           ((null (setq id-token (parse-identifier))) nil)
           ((null (parse-r-paren)) nil)
           ((cdr id-token)))))


(defun parse-args ()
   (let ((id-token) (al-token))
     (cond ((null (parse-l-paren)) nil)
           ((null (setq id-token (parse-identifier))) nil)
           ((null (setq al-token (parse-arg-list)))
            (cond ((null (parse-r-paren)) nil)
                  ((cdr id-token))))
           ((cond ((null (parse-r-paren)) nil)
                  ((append (cdr id-token) al-token)))))))


(defun parse-arg-list ()
   (let ((id-token) (al-token))
        (cond ((null (parse-comma)) nil)
              ((null (setq id-token (parse-identifier))) nil)
              ((append (cdr id-token) (parse-arg-list))))))
```

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;                                                                  ;;
;;             ***** parse-conceptual-graph *****                   ;;
;;                                                                  ;;
;; This function parses a syntactically legal conceptual graph.     ;;
;;                                                                  ;;
;; If a syntactically legal conceptual graph is found, returns      ;;
;;                                                                  ;;
;;   (a list containing                                             ;;
;;            "conceptual-graph" and                                ;;
;;            any number occurances of                              ;;
;;              (list containing                                    ;;
;;                  conceptual-relation                             ;;
;;                  from-concept                                    ;;
;;                  to-concept))                                    ;;
;; Else returns nil                                                 ;;
;;                                                                  ;;
;; This essentially converts the graphical form of a conceptual     ;;
;; graph (having "repeating groups") into a flat list structure.    ;;
;;                                                                  ;;
;; For example, given the graph                                     ;;
;;                                                                  ;;
;;      [SIT] -                                                     ;;
;;         (AGNT) -> [CAT]                                          ;;
;;         (LOC)  -> [MAT]                                          ;;
;;         (PTIM) -> [DAY : Yesterday]                              ;;
;;                                                                  ;;
;;    this function would return                                    ;;
;;                                                                  ;;
;;           (conceptual-graph                                      ;;
;;              (AGNT (SIT) (CAT))                                  ;;
;;              (LOC  (SIT) (MAT))                                  ;;
;;              (PTIM (SIT) (DAY (label Yesterday))))               ;;
;;                                                                  ;;
;;  Note that the "to-concept" of the last list has a referent,     ;;
;;  which is the label "Yesterday".  Both "from-concepts" and       ;;
;;  to-concepts can be base-concept - referent pairs.  The          ;;
;;  different kinds of referents are documented in the code         ;;
;;  preceding the function "parse-referent".                        ;;
;;                                                                  ;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
```

```
(defun parse-conceptual-graph ()
   (let ((right-linked-relation) (cnode) (lc))
     (setq right-linked-relation (parse-right-linked-relation))
     (cond ((null (setq cnode (parse-cnode))) nil)
           ((list 'conceptual-graph
               (cond ((null right-linked-relation)
                      (cond ((parse-linked-concept cnode))
                         ((list cnode))))
                     ((cond ((setq lc (parse-linked-concept cnode))
                             (append (list
                               (list
                                  right-linked-relation
                                  cnode))
                                     lc))
                      ((list
                          right-linked-relation
                          cnode)))))))))


(defun parse-linked-concept (concept)
   (cond ((parse-right-linked-concept concept))
      ((parse-left-linked-concept concept))
      ((parse-bi-dir-concept-list concept))))


(defun parse-right-linked-concept (concept)
   (let ((ral))
     (cond ((null (parse-right-arc)) nil)
           ((null (setq ral (parse-right-arc-link concept))) nil)
           ((append (list (list (car ral)
                                concept
                          (caddr ral)))
                 (parse-linked-concept (caddr ral)))))))


(defun parse-left-linked-concept (concept)
   (let ((lal))
     (cond ((null (parse-left-arc)) nil)
           ((null (setq lal (parse-left-arc-link concept))) nil)
           ((append (list (list (car lal)
                                concept
                          (cadr lal)))
                 (parse-linked-concept (caddr lal)))))))
```

```
(defun parse-left-linked-relation (to-concept)
   (let ((relation))
        (cond ((null (setq relation (parse-relnode))) nil)
           ((list relation to-concept)))))


(defun parse-right-linked-relation ()
   (let ((relation))
     (cond ((null (setq relation (parse-relnode))) nil)
           ((null (parse-right-arc)) (unget-token (list relation)))
           (relation))))


(defun parse-bi-dir-concept-list (concept)
   (let ((cl))
     (cond ((null (parse-bidirectional-arc)) nil)
           ((null (setq cl (parse-concept-list concept))) nil)
           ((null (parse-rel-list-terminator)) nil)
           (t cl))))


(defun parse-concept-list (concept)
   (let ((arc-link) (left-linked-relation))
     (cond ((setq arc-link (parse-arc-link concept))
            (append arc-link (parse-concept-list concept)))
           ((setq left-linked-relation
              (parse-left-linked-relation concept))
            (append (list left-linked-relation)
                 (parse-concept-list concept))))))


(defun parse-arc-link (concept)
   (let ((right) (left))
        (cond ((setq right (parse-right-arc-link concept))
            (append (list right) (parse-linked-concept
                              (caddr right))))
              ((setq left (parse-left-arc-link concept))
            (append (list left)
                 (parse-linked-concept (caddr left)))))))


(defun parse-right-arc-link (from-concept)
   (let ((relation) (to-concept))
     (cond ((null (setq relation (parse-relnode))) nil)
           ((null (parse-right-arc))
            (unget-token (list (list relation))))
           ((null (setq to-concept (parse-cnode)))
           (unget-token '(->))
           (unget-token (list (list relation))))
           ((list relation from-concept to-concept)))))
```

```
(defun parse-left-arc-link (to-concept)
   (let ((relation) (from-concept))
     (cond ((null (setq relation (parse-relnode))) nil)
           ((null (parse-left-arc))
           (unget-token (list (list relation))))
           ((null (setq from-concept (parse-cnode))) nil)
           ((list relation from-concept to-concept)))))


(defun parse-cnode ()
   (let ((bc-token) (ref-token))
     (cond ((null (parse-l-bracket)) nil)
           ((null (setq bc-token (parse-base-concept))) nil)
           ((parse-r-bracket)  bc-token)
           ((parse-colon)
            (cond ((null (setq ref-token (parse-referent))) nil)
               ((null (parse-r-bracket)) nil)
               (t  (append bc-token ref-token)))))))


(defun parse-relnode ()
   (let ((id-token))
     (cond ((null (parse-l-paren)) nil)
           ((null (setq id-token (parse-identifier))) nil)
           ((null (parse-r-paren)) nil)
           ((cadr id-token)))))


(defun parse-rel-list-terminator ()
   (cond ((null INPUT))
      ((null (parse-comma)) nil)
      ((null INPUT) nil)
      (t t)))


(defun parse-base-concept ()
   (let ((id-token) (string-token))
     (cond ((setq id-token (parse-identifier))
            (cdr id-token))
              ((setq string-token (parse-string))
            (cdr string-token)))))
```

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;                                                                              ;;
;;                  ***** parse-referent *****                                  ;;
;;                                                                              ;;
;;  This function parses a syntactically legal referent.                        ;;
;;                                                                              ;;
;;     The types of referents are specified in the code                         ;;
;;     for this function.                                                       ;;
;;                                                                              ;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(defun parse-referent ()
   (let ((token))
        (cond
           ((setq token (parse-set-description))
               token)
           ((setq token (parse-identifier))
            (append '(label) (cdr token)))
           ((setq token (parse-number))
            (list 'label token))
           ((setq token (parse-individual-marker))
            token)
           ((setq token (parse-variable))
            token)
           ((setq token (parse-generic-individual-marker))
            token)
           ((setq token (parse-contracted-measure))
            token)
           ((setq token (parse-conceptual-graph))
            token))))


(defun parse-individual-marker ()
   (let ((sharp-sign-token) (number-token))
     (cond ((null (setq sharp-sign-token (parse-sharp-sign))) nil)
           ((null (setq number-token (parse-integer))) nil)
           ((list 'individual-marker number-token)))))


(defun parse-generic-individual-marker ()
   (let ((token))
        (cond ((null (setq token (parse-asterisk))) nil)
           ((list 'generic-individual (car token))))))
```

```
(defun parse-contracted-measure ()
   (let ((at-token) (number-token) (units-token))
        (cond ((null (setq at-token (parse-at-sign))) nil)
              ((null (setq number-token (parse-number))) nil)
              ((null (setq units-token (parse-units))) nil)
              ((list 'contracted-measure
                  (list number-token (car units-token)))))))


(defun parse-variable ()
   (let ((asterisk-token) (variable-token))
     (cond ((null (setq asterisk-token (parse-asterisk))) nil)
           ((null (setq variable-token (parse-identifier)))
            (unget-token asterisk-token))
           ((append '(variable) (cdr variable-token))))))


(defun parse-units ()
   (let ((units (parse-identifier)))
     (cond ((null (car units)) nil)
           (t (cdr units)))))


(defun parse-set-description ()
   (cond ((parse-respective-set))
      ((parse-distributive-set))
      ((parse-disjunctive-set))
      ((parse-collective-set))))


(defun parse-distributive-set ()
   (let ((id-token) (cj-clause))
        (cond ((null (setq id-token (parse-identifier))) nil)
              ((not (equal (cadr id-token) 'Dist)) (unget-token id-token))
              ((null (setq cj-clause (parse-conjunctive-clause)))
                    (unget-token id-token))
              ((append '(distributive-set) cj-clause)))))


(defun parse-collective-set ()
   (let ((conjunct) (set-element))
     (cond
           ((setq conjunct (parse-conjunctive-clause))
           (append '(collective-set) conjunct))
           ((null (parse-l-brace)) nil)
           ((null (setq set-element (parse-set-element)))
            (unget-token '(\{)))
           ((null (parse-r-brace))
            (unget-token set-element)
            (unget-token '(\{)))
           ((append '(collective-set) set-element)))))
```

```
(defun parse-disjunctive-set ()
   (let ((token (parse-disjunctive-clause)))
        (cond ((null token) nil)
           (t (append '(disjunctive-set) token)))))


(defun parse-respective-set ()
   (let ((token (parse-respective-clause)))
        (cond ((null token) nil)
           (t (append '(respective-set) token)))))


(defun parse-conjunctive-clause ()
   (let ( (cjs) (card) )
        (cond
           ((null (parse-l-brace)) nil)
           ((parse-asterisk)
                 (cond ((null (parse-r-brace))
                     (unget-token '(\{)))
                    (t '(\*))))
           ((null (setq cjs (parse-conjunctive-sequence)))
               (unget-token '(\{)))
           ((null (parse-generic-addition))
            (cond ((null (parse-r-brace)) nil)
               (t (cdr cjs))))
           ((null (parse-r-brace)) nil)
           ((null (setq card (parse-cardinality)))
              (append (cdr cjs) '(*) ))
           ((append (cdr cjs) '(*) (list card))))))


(defun parse-disjunctive-clause ()
   (cond ((null (parse-l-brace)) nil)
      ((let ((djs (parse-disjunctive-sequence)))
            (cond ((null djs) (unget-token '(\{)))
               ((null (parse-r-brace)) nil)
               (t (cdr djs)))))))


(defun parse-respective-clause ()
   (let ((id-token) (cjs))
     (cond ((null (setq id-token (parse-identifier))) nil)
           ((not (equal (cadr id-token) 'Resp)) (unget-token id-token))
           ((null (parse-l-angle-bracket)) (unget-token id-token))
           ((null (setq cjs (parse-conjunctive-sequence))) nil)
           ((null (parse-r-angle-bracket)) nil)
           ((cdr cjs)))))
```

```
(defun parse-disjunctive-sequence ()
   (let ((id-token) (dsl-token))
     (cond ((null (setq id-token (parse-identifier))) nil)
           ((null (setq dsl-token
                  (parse-disjunctive-sequence-list)))
            (unget-token id-token))
           ((append '(disjunctive-sequence)
                 (cdr id-token)
                 dsl-token)))))


(defun parse-disjunctive-sequence-list ()
   (let ((id-token) (dsl-token))
        (cond ((null (parse-vertical-bar)) nil)
              ((null (setq id-token (parse-identifier)))
               (unget-token '(\|)))
              ((append (cdr id-token)
                    (parse-disjunctive-sequence-list))))))


(defun parse-conjunctive-sequence ()
   (let ((id-token))
     (cond ((null (setq id-token (parse-identifier))) nil)
           ((append '(conjunctive-sequence)
                 (cdr id-token)
                 (parse-conjunctive-sequence-list))))))


(defun parse-conjunctive-sequence-list ()
   (let ((id-token))
     (cond ((null (parse-comma)) nil)
           ((null (setq id-token (parse-identifier)))
            (unget-token '(\,)))
           ((append (cdr id-token)
                 (parse-conjunctive-sequence-list))))))


(defun parse-generic-addition ()
   (cond ((parse-comma)
          (cond ((parse-asterisk) '(*))
                ((unget-token '(\,)))))))


(defun parse-cardinality ()
   (let ((card))
     (cond ((null (parse-at-sign)) nil)
           ((null (setq card (parse-integer)))
           (unget-token '(\@)))
           ((list 'cardinality card)))))
```

```
(defun parse-set-element ()
   (let ((token (parse-identifier)))
        (cond (token token)
           ((parse-asterisk)))))


(defun parse-real-number ()
   (let ((whole) (fraction))
     (cond ((setq whole (parse-integer))
            (cond ((null (parse-period)) (unget-token (list whole)))
               ((setq fraction (parse-integer))
                   (implode (append (explode whole)
                                    '(\.)
                                    (explode fraction))))
               ((implode (append (explode whole) '(\.))))))
           ((null (parse-period)) (unget-token (list whole)))
           ((setq fraction (parse-integer))
            (implode (append '(\.) (explode fraction)))))))


;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;                                                                           ;;;
;;;          ****** Token Parsing Functions ********                          ;;;
;;;                                                                           ;;;
;;;      Each of the following parsing functions parse for                    ;;;
;;;   a single token returned by the lexical scanner.  Their                  ;;;
;;;   actions are all the same, except for the identity of the                ;;;
;;;   token being parsed for:                                                 ;;;
;;;                                                                           ;;;
;;;       -- Get a token from the input stream                                ;;;
;;;             (by a call to 'next-token')                                   ;;;
;;;                                                                           ;;;
;;;       -- If the token is the one being parsed for                         ;;;
;;;                                                                           ;;;
;;;             then  return the token as the value of the                    ;;;
;;;                   parsing function                                        ;;;
;;;                                                                           ;;;
;;;             else  put the token back on the input stream                  ;;;
;;;                   (by a call to 'unget-token' and return                  ;;;
;;;                   nil as the value of the parsing function                ;;;
;;;                                                                           ;;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;


(defun parse-string ()
   (let ((token (next-token)))
     (cond ((equal (car token) 'string) token)
           (t (unget-token token)))))
```

```
(defun parse-identifier ()
   (let ((token (next-token)))
     (cond ((equal (car token) 'identifier) token)
           (t (unget-token token)))))


(defun parse-number ()
   (cond ((parse-real-number))
      ((parse-integer))))


(defun parse-integer ()
   (let ((token (next-token)))
     (cond ((equal (car token) 'integer) (cadr token))
           (t (unget-token token)))))


(defun parse-l-paren ()
   (let ((token (next-token)))
        (cond ((equal (car token) '\( ) token)
           (t (unget-token token)))))

(defun parse-r-paren ()
   (let ((token (next-token)))
        (cond ((equal (car token) '\) ) token)
           (t (unget-token token)))))


(defun parse-l-bracket ()
   (let ((token (next-token)))
        (cond ((equal (car token) '\[ ) token)
           (t (unget-token token)))))

(defun parse-r-bracket ()
   (let ((token (next-token)))
        (cond ((equal (car token) '\] ) token)
           (t (unget-token token)))))

(defun parse-l-brace ()
   (let ((token (next-token)))
        (cond ((equal (car token) '\{ ) token)
           (t (unget-token token)))))

(defun parse-r-brace ()
   (let ((token (next-token)))
        (cond ((equal (car token) '\} ) token)
           (t (unget-token token)))))
```

```
(defun parse-colon ()
   (let ((token (next-token)))
        (cond ((equal (car token) '\: ) token)
           (t (unget-token token)))))

(defun parse-comma ()
   (let ((token (next-token)))
        (cond ((equal (car token) '\, ) token)
           (t (unget-token token)))))

(defun parse-period ()
   (let ((token (next-token)))
        (cond ((equal (car token) '\. ) token)
           (t (unget-token token)))))

(defun parse-sharp-sign ()
   (let ((token (next-token)))
        (cond ((equal (car token) '\# ) token)
           (t (unget-token token)))))

(defun parse-at-sign ()
   (let ((token (next-token)))
        (cond ((equal (car token) '\@ ) token)
           (t (unget-token token)))))

(defun parse-asterisk ()
   (let ((token (next-token)))
        (cond ((equal (car token) '\* ) token)
           (t (unget-token token)))))

(defun parse-vertical-bar ()
   (let ((token (next-token)))
        (cond ((equal (car token) '\| ) token)
           (t (unget-token token)))))

(defun parse-l-angle-bracket ()
   (let ((token (next-token)))
        (cond ((equal (car token) '\< ) token)
           (t (unget-token token)))))


(defun parse-r-angle-bracket ()
   (let ((token (next-token)))
        (cond ((equal (car token) '\> ) token)
           (t (unget-token token)))))

(defun parse-left-arc ()
   (let ((token (next-token)))
        (cond ((equal (car token) '\<- ) token)
           (t (unget-token token)))))
```

```
(defun parse-right-arc ()
   (let ((token (next-token)))
        (cond ((equal (car token) '\-> ) token)
           (t (unget-token token)))))

(defun parse-bidirectional-arc ()
   (let ((token (next-token)))
        (cond ((equal (car token) '\- ) token)
           (t (unget-token token)))))

(defun makeinput (in)
   (setq INPUT (reverse (cdr (reverse (cdr (explode in)))))))
```

Appendix D : Example of Use of the Tools

This appendix contains scripts of examples of the use of the tools for processing the definition of types, individuals, conceptual relations, schemata and prototypes.

To use these tools, first start Franz Lisp (type 'lisp'), or PEARL (type 'pearl'), and load a file containing the tools tools. The uncompiled version in /usrb/src/ksu/sowa/src/sowa.l and the compiled version in /usrb/src/ksu/sowa/bin/sowa.o produce the same results, although the compilee version runs approximately ten times faster than the interpreted version.

Once one of these files has been loaded, all the parsing functions for conceptual graphs (as listed in Appendix C) are now available. They can be invoked from the top level by typing in a definition from the Lisp '-> ' or PEARL 'pearl> ' prompt, or definitions can be made by loading one or more files containing definitions.

The remainder of this appendix contains examples in the following format:

-- PEARL is involed from UNIX level (the prompt 'pearl> ' is now displayed

-- The file /usrb/src/ksu/sowa/bin/sowa.o is loaded
(pearl> (load '/usrb/src/ksu/sowa/bin/sowa.o))

-- The contents of the file containing the definition(s) to be made is displayed (pearl> (exec cat <filename>))

-- The file containing the definition is loaded
(pearl> (load '<filename>))

-- The property list containing the processed form of the definition is displayed
(pearl> (get '<dbname> '<definition-name>))

-- PEARL is exited by typing ^D, returning to UNIX

```
% pearl
Franz Lisp, Opus 38.79 plus PEARL 3.9
pearl> (load '/usrb/src/ksu/sowa/bin/sowa.o)
[fasl /usrb/src/ksu/sowa/bin/sowa.o]
   t
pearl> (exec cat type)
(type AI-TEACHER (x) is
   [TEACH] -
      (AGNT) -> [TEACHER : *x]
      (OBJ)  -> [SUBJECT : AI]
      (LOC)  -> [UNIVERSITY])

pearl> (load 'type)
[load type]
   t
pearl> (get 'TYPE 'AI-TEACHER)
   ((x)
    (conceptual-graph
     ((AGNT (TEACH) (TEACHER variable x))
      (OBJ (TEACH) (SUBJECT label AI))
      (LOC (TEACH) (UNIVERSITY)))))
pearl> ^D
Goodbye
%
```

```
% pearl
Franz Lisp, Opus 38.79 plus PEARL 3.9
pearl> (load '/usrb/src/ksu/sowa/bin/sowa.o)
[fasl /usrb/src/ksu/sowa/bin/sowa.o]
   t
pearl> (exec cat individual)
(individual AI-TEACHER (rth) is
   (PAST) -> [TEACH] -
               (AGNT) -> [TEACHER : rth]
               (OBJ)  -> [SUBJECT : AI]
               (LOC)  -> [UNIVERSITY : KSU])
   0
pearl> (load 'individual)
[load individual]
   t
pearl> (get 'INDIVIDUAL 'rth)
   ((identifier AI-TEACHER)
    (conceptual-graph
     ((PAST (TEACH))
      (AGNT (TEACH) (TEACHER label rth))
      (OBJ (TEACH) (SUBJECT label AI))
      (LOC (TEACH) (UNIVERSITY label KSU)))))
pearl> ^D
Goodbye
%
```

```
% pearl
Franz Lisp, Opus 38.79 plus PEARL 3.9
pearl> (load '/usrb/src/ksu/sowa/bin/sowa.o)
[fasl /usrb/src/ksu/sowa/bin/sowa.o]
   t
pearl> (exec cat relation)

(relation ABOVE (x,y) is
   [POSITION : *x ] -
       (CHRC) -> [Y-CORD : *a ]-
                    (GT) -> [Y-CORD : *b ] -
                                (CHRC) <- [POSITION : *y ])

(relation BELOW (x,y) is
   [POSITION : *x ] -
       (CHRC) -> [Y-COORD : *a ]-
                    (LT) -> [Y-COORD : *b ] -
                                (CHRC) <- [POSITION : *y ])

(relation ON (x,y) is
   [OBJECT : *x ] -
      (ABUT) -> [OBJECT : *y]
      (CHRC) -> [POSITION : *a ] -
                    (CHRC) -> [Y-COORD : *b ]-
                                 (GT) -> [Y-COORD : *c ] -
                                             (CHRC) <- [POSITION : *d ] -
                                                          (CHRC) <-
                                                            [OBJECT :*y ])
   0
pearl> (load 'relation)
[load relation]
   t
pearl> (get 'RELATION 'ABOVE)
   ((x y)
    (conceptual-graph
     ((CHRC (POSITION variable x) (Y-CORD variable a))
      (GT (Y-CORD variable a) (Y-CORD variable b))
      (CHRC (POSITION variable y) (Y-CORD variable b)))))
pearl> (get 'RELATION 'BELOW)
   ((x y)
    (conceptual-graph
     ((CHRC (POSITION variable x) (Y-COORD variable a))
      (LT (Y-COORD variable a) (Y-COORD variable b))
      (CHRC (POSITION variable y) (Y-COORD variable b)))))
pearl> (get 'RELATION 'ON)
   ((x y)
    (conceptual-graph
     ((ABUT (OBJECT variable x) (OBJECT variable y))
      (CHRC (OBJECT variable x) (POSITION variable a))
      (CHRC (POSITION variable a) (Y-COORD variable b))
      (GT (Y-COORD variable b) (Y-COORD variable c))
      (CHRC (POSITION variable d) (Y-COORD variable c))
      (CHRC (OBJECT variable y) (Y-COORD variable c)))))
pearl> ^D
Goodbye
%
```

```
% pearl
Franz Lisp, Opus 38.79 plus PEARL 3.9
pearl> (load '/usrb/src/ksu/sowa/bin/sowa.o)
[fasl /usrb/src/ksu/sowa/bin/sowa.o]
   t
pearl> (exec cat prototype1)
(prototype for CAT (x) is
   [PET: *x] -
      (CHRC)      -> [NAME : Tom]
      (CHRC)      -> [AGE : @ 5 years]
      (OWNED-BY) -> [OWNER])
   0
pearl> (load 'prototype1)
[load prototype1]
   t
pearl> (get 'PROTOTYPE 'CAT)
   (((x)
     (conceptual-graph
      ((CHRC (PET variable x) (NAME label Tom))
       (CHRC (PET variable x) (AGE contracted-measure (\5 years)))
       (OWNED-BY (PET variable x) (OWNER))))))
pearl> (exec cat prototype2)
(prototype for CAT (x) is
  [MAMMAL: *x] -
    (CHRC) -> [GENUS : Felis]
    (CHRC) -> [SPECIES : Domesticus])
   0
pearl> (load 'prototype2)
[load prototype2]
   t
pearl> (get 'PROTOTYPE 'CAT)
   (((x)
     (conceptual-graph
      ((CHRC (PET variable x) (NAME label Tom))
       (CHRC (PET variable x) (AGE contracted-measure (\5 years)))
       (OWNED-BY (PET variable x) (OWNER)))))
    ((x)
     (conceptual-graph
      ((CHRC (MAMMAL variable x) (GENUS label Felis))
       (CHRC (MAMMAL variable x) (SPECIES label Domesticus))))))
pearl> ^D
Goodbye
5.0u 4.4s 2:24 6% 103+351k 61+27io 395pf+0w
%
```

```
% pearl
Franz Lisp, Opus 38.79 plus PEARL 3.9
pearl> (load '/usrb/src/ksu/sowa/bin/sowa.o)
[fasl /usrb/src/ksu/sowa/bin/sowa.o]
   t
pearl> (exec cat schema1)
(schema for CAT (x) is
   [PET: *x] -
      (CHRC) -> [AGE]
      (OWNED-BY) -> [OWNER])
   0
pearl> (load 'schema1)
[load schema1]
   t
pearl> (get 'SCHEMA 'CAT)
   (((x)
     (conceptual-graph
      ((CHRC (PET variable x) (AGE))
       (OWNED-BY (PET variable x) (OWNER))))))
pearl> (exec cat schema2)
(schema for CAT (x) is
  [MAMMAL: *x] -
    (CHRC) -> [NAME]
    (CHRC) -> [GENUS : Felis]
    (CHRC) -> [SPECIES])
   0
pearl> (load 'schema2)
[load schema2]
   t
pearl> (get 'SCHEMA 'CAT)
   (((x)
     (conceptual-graph
      ((CHRC (PET variable x) (AGE))
       (OWNED-BY (PET variable x) (OWNER)))))
    ((x)
     (conceptual-graph
      ((CHRC (MAMMAL variable x) (NAME))
       (CHRC (MAMMAL variable x) (GENUS label Felis))
       (CHRC (MAMMAL variable x) (SPECIES))))))
pearl> ^D
Goodbye
%
```

Bibliography

Aho, Alfred V. & Jeffrey D. Ullman, "Principles of Compiler Design", Bell Telephone Laboratories, Inc., 1977.

Aristotle, "The Categories, On Interpretation, Prior Analytics, Posterior Analytics, Topica", Loeb Classical Library, Harvard University Press, Cambridge, MA.

Deering, Michael, Joseph Faletti, & Robert Wilensky, "Using the PEARL AI Package (Package for Efficient Access to Representations in Lisp), Computer Science Division, Department of EECS, University of California, Berkeley, 1982.

Hartley, Roger T., "Representation of Procedural Knowledge for Expert Systems", Unpublished Manuscript, Kansas State University, Manhattan, 1985.

Newell, Allen & Herbert A. Simon, "Computer Science as Empirical Inquiry: Symbols and Search", Communications of the ACM, Vol. 19, No. 3, Mar. 1976.

Rich, Elaine, "Artificial Intelligence", McGraw-Hill, New York, NY, 1983.

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

White, Alan R., "Conceptual analysis," in C.J. Botempo & S.J. Odell, eds., "The Owl of Minerva", McGraw-Hill, New York, NY 1975.

Winston, Patrick Henry, "Artificial Intelligence", Addison-Wesley, Reading, MA, 1984.

ACKNOWLEDGEMENTS

There were many people who helped to make my Master's program a success; I would like to extend my gratitude to these people.

First, I would like to thank my major professor, Dr. Elizabeth Unger. Beth's assistance with the project and in helping me find my niche in the computer science field has given me faith that an ex-Wildlife Biologist can be happy in a field other than ecology.

I would also like to thank my graduate committee, Dr. Richard McBride and Dr. Austin Melton.

Thanks also to Dr. Roger Hartley for initially exposing me to the field of artificial intelligence.

I also owe a debt of gratitude to Harvard Townsend for his understanding in times when the need to work on my project (especially this final semester) and classes, all in four days a week, sometimes conflicted with my work schedule.

Finally, and most importantly, I want to thank my fiance Karen Holmes (who will be my wife from December 22 on) for putting up with a weekend relationship for over a year. The motivation she provided more than made up for the three days a week when I wasn't working on academic activities. The MS degree belongs to her as well as to me; I scarcely could have done it without her understanding and support.

Representation of Knowledge using Sowa's Conceptual Graphs: An Implementation of a Set of Tools

by

Gary Schiltz

B.S., Kansas State University, 1985

AN ABSTRACT OF A MASTER'S REPORT

Submitted in Partial Fulfillment of the

Requirements for the Degree

MASTER OF SCIENCE

Department of Computer Science

KANSAS STATE UNIVERSITY
Manhattan, Kansas

1986

ABSTRACT

Representation of real world knowledge on a digital computer is a key issue in modern computer science. The theory of conceptual graphs has risen to prominence among knowledge representation schemes; there has been, however, no implementation in public domain of automated tools for representing knowledge using the linear form of conceptual graphs. This paper introduces the importance of knowledge representation, provides a short introduction to conceptual graph theory and outlines the implementation of such a set of tools.