Semantic Analysis for System Level Design Automation

by

Rob Greenwood

Thesis submitted to the Faculty of the

Virginia Polytechnic Institute and State University

in partial fulfillment of the requirements for the degree of

Master of Science

in

Electrical Engineering

APPROVED:

[Signature]

Dr. Walling R. Cyre

[Signature]

Dr. James R. Armstrong

[Signature]

Dr. Dong S. Ha

September, 1992

Blacksburg, Virginia
Semantic Analysis for System Level Design Automation

by

Rob Greenwood

Dr. Walling R. Cyre, Chairman

Electrical Engineering

(ABSTRACT)

This thesis describes the design and implementation of a system to extract meaning from natural language specifications of digital systems. This research is part of the ASPIN project which has the long-term goal of providing an automated system for digital system synthesis from informal specifications.

This work makes several contributions, one being the application of artificial intelligence techniques to specifications writing. Also, the work deals with the subset of the English language used to describe digital systems, and the concepts within this domain have been classified into a type hierarchy. Finally, a set of relations has been defined to represent the interrelationships between the concepts of the sublanguage.

This work centers around the modeling of information found in natural language specifications of digital systems. The target knowledge representation for the work is the conceptual graph, developed by John Sowa. Conceptual graphs provide a sound theoretical base as well as enough versatility to model the information found in digital system specifications. The transformation from natural language to conceptual graphs is done in two stages. In the first stage, a previously developed context-free English language parser is used to create trees of sentence structure. In the second stage, the trees
are processed by a semantic analyzer which uses a conceptual type hierarchy and a database of rules to extract the meaning from the English sentence and create the conceptual graph.

The main work of this thesis centers around the semantic analyzer which is written in Quintus Prolog. The semantic analyzer currently contains approximately 380 canonical conceptual graphs that cover usage of over 680 words consisting of over 240 nouns and over 460 verbs.
Acknowledgements

I would like to thank Dr. Walling Cyre for the patience and support he showed throughout the duration of this project. I am especially grateful for the numerous ideas and suggestions he provided that helped me get through some of the problems that inevitably occur in natural language processing systems. Also, I would like to thank him for his attention to detail that often opened my eyes to mistakes that I had overlooked. I would also like to thank Dr. James Armstrong and Dr. Dong Ha for serving on my committee. In addition, I thank Semiconductor Research Corporation for the support and funding they provided for this project.

I am grateful to Sandeep Shah, Chang Cho, Bi-Yu Pan, Ming Chu, and others in the lab who were unselfish in their help to me and provided encouragement to finish this work. I am especially grateful to my wife Susan who has been especially understanding and encouraging these last few weeks. Her friendship provided me the strength to complete this project. Finally, I would like to thank God for providing so much joy in my life and for always watching out for me.
# Table of Contents

**Chapter 1. Introduction** .................................................. 1

1.1. Motivation ................................................................. 1

1.2. The ASPIN System ..................................................... 3

1.3. Research Approach ................................................... 6

1.4. Thesis Organization .................................................. 7

**Chapter 2. Background** .................................................. 9

2.1. Review of English Language Terminology .......................... 9

2.2. Introduction to Linguistics ........................................... 12

2.3. Related Research ....................................................... 17

2.3.1. PHRAN-SPAN System ............................................. 17

2.3.2. Sowa and Way's PLNLP System .................................. 20

2.3.3. Pazienza and Velardi's Natural Language Understanding System .... 22

**Chapter 3. Conceptual Graphs** ...................................... 26

3.1. Introduction ............................................................. 26
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.2.</td>
<td>Semantic Model</td>
<td>30</td>
</tr>
<tr>
<td>4.</td>
<td>Chapter 4. Syntactic Processing</td>
<td>35</td>
</tr>
<tr>
<td>4.1.</td>
<td>Introduction</td>
<td>35</td>
</tr>
<tr>
<td>4.2.</td>
<td>ASPIN Parser</td>
<td>39</td>
</tr>
<tr>
<td>5.</td>
<td>Chapter 5. Semantic Analysis</td>
<td>45</td>
</tr>
<tr>
<td>5.1.</td>
<td>Introduction</td>
<td>45</td>
</tr>
<tr>
<td>5.2.</td>
<td>Concept Type Hierarchy</td>
<td>47</td>
</tr>
<tr>
<td>5.3.</td>
<td>Canonical Conceptual Graphs</td>
<td>50</td>
</tr>
<tr>
<td>5.4.</td>
<td>Role Markers</td>
<td>52</td>
</tr>
<tr>
<td>5.5.</td>
<td>Attach Operation</td>
<td>53</td>
</tr>
<tr>
<td>5.6.</td>
<td>Semantic Rules</td>
<td>57</td>
</tr>
<tr>
<td>5.6.1.</td>
<td>Adjective Rules</td>
<td>58</td>
</tr>
<tr>
<td>5.6.2.</td>
<td>Adverbial Rules</td>
<td>59</td>
</tr>
<tr>
<td>5.6.3.</td>
<td>Head Rules</td>
<td>60</td>
</tr>
<tr>
<td>5.6.4.</td>
<td>Nominal Rules</td>
<td>64</td>
</tr>
<tr>
<td>5.6.5.</td>
<td>Predicate Rules</td>
<td>66</td>
</tr>
<tr>
<td>5.6.6.</td>
<td>Conjunction Processing</td>
<td>67</td>
</tr>
<tr>
<td>5.7.</td>
<td>Inferencing</td>
<td>69</td>
</tr>
<tr>
<td>6.</td>
<td>Chapter 6. Analysis Examples</td>
<td>71</td>
</tr>
<tr>
<td>6.1.</td>
<td>Introduction</td>
<td>71</td>
</tr>
<tr>
<td>6.2.</td>
<td>A Detailed Example</td>
<td>71</td>
</tr>
<tr>
<td>6.3.</td>
<td>Other Analysis Examples</td>
<td>77</td>
</tr>
</tbody>
</table>
List of Figures

Figure 1. ASPIN System Overview ........................................ 4
Figure 2. ASPIN System Organization ................................. 5
Figure 3. Grammatical Structure Tree for *The register is reset* .......... 14
Figure 4. Graphical Representation of a Conceptual Graph ............. 27
Figure 5. Conceptual Graph for *The processor reads the data from memory* .. 29
Figure 6. Canonical Graphs of "load" and "microcontroller" .......... 32
Figure 7. Canonical Graph of "load" After Restriction .................. 32
Figure 8. Join of Canonical Graphs in Figure 6 .......................... 33
Figure 9. Structure of the Conceptual Type Hierarchy .................. 34
Figure 10. Example Grammar Rules Used by the ASPIN Parser .......... 36
Figure 11. Simplified Parse Chart for "The signal resets the register" .... 38
Figure 12. Parse Chart for "the" ........................................ 40
Figure 13. Parse Chart for "the signal" .................................. 41
Figure 14. Parse Tree of "the signal resets the register" ................ 41
Figure 15. Two Parse Trees for "the input signals reset" ................ 42
Figure 16. Parse Chart for "a constant executes the bus" ............... 43
Figure 17. Semantic Analyzer Organization ............................ 46
Figure 18. Partial Conceptual Type Hierarchy ........................................... 48
Figure 19. Flow Chart of the Attach Operation ................................. 55
Figure 20. Canonical Graph for the Adjective "8-bit" ......................... 59
Figure 21. Identifier Attachment ....................................................... 62
Figure 22. Syntactic Parse Tree for Sentence #1 ............................... 73
Figure 23. Canonical Graphs for "load," "data," "rise," and "8-bit" .......... 74
Figure 24. Conceptual Graph of Sentence #1 After np11 is Processed ...... 74
Figure 25. Conceptual Graph of Sentence #1 After pvs1 is Processed ...... 75
Figure 26. Conceptual Graph of Sentence #1 After Subject Attachment .... 76
Figure 27. Complete Conceptual Graph of Sentence #1 ....................... 77
List of Tables

Table 1. Types of English Sentences .................................. 12
Table 2. Definitions of Fillmore's Cases ............................... 16
Table 3. Definitions of the Most General Conceptual Types .......... 47
Table 4. Examples of Conceptual Subtypes ............................ 49
Table 5. Grammatical Categories for the Words in Sentence #1 ....... 72
Table 6. ASPIN Grammar Rules ....................................... 101
Chapter 1. Introduction

1.1. Motivation

Design cycle times for digital systems have been greatly reduced due to the progress made in the area of synthesis. Most synthesis systems rely on some sort of formal representation, such as hardware description languages, to derive requirements information. Formal models, however, are not usually available during the initial design stages. In practice, design of digital systems usually begins with the development of a set of specifications outlining the requirements for the desired system. These specifications usually consist of various forms of information, such as block diagrams, timing diagrams, flow charts, and natural language. Formal models for synthesis must be manually created from these informal design notations. Of these design notations, natural language is the least formal, yet the most versatile for describing system behavior and characteristics.

Most of the difficulty encountered in natural language specifications is due to the richness of the English language. Different interpretations can be drawn from the same
set of specifications. Therefore, the interpretations of a reader may be different than that of the specifications author. This can result in costly design errors.

Several types of flaws are inherent in specifications written in natural language [4]. First, there are no means to determine if the system description is complete enough to produce a correct implementation of the system. The specifications author might assume that some vital information is known by the reader. Still other information could simply be omitted by mistake. Secondly, some information in the specification may be incorrect. There are two types of inconsistencies that usually occur: syntactic inconsistencies and semantic inconsistencies. Syntactic inconsistencies deal with such things as the names of items and number of inputs. Semantic inconsistencies are related to the interpretations of the behavior of the system. For example, two states that are identical in a system might be described as having different behaviors. These type of contradictions are the most difficult to detect. Syntactic contradictions in requirements might be able to be detected through extensive analysis, but without some other high level specification to compare with, incorrect information is difficult to detect. Other problems can result due to over specification of the requirements. This results in aspects of the system that are overly constrained resulting from non-essential information.

Another issue of importance in system design is back-annotation. During design, it is common that implementation restrictions make it impossible to follow initial design constraints. When this occurs, it becomes necessary to modify the initial requirements in order for the specifications to accurately model the system. Due to time pressures and difficulty in tracing the source of requirements, this is often left undone. This results in specifications that do not accurately represent the system.
An automated system for specifications development would help eliminate some of the problems mentioned above. Such a system would standardize the interpretation of natural language specifications and eliminate much of the ambiguity at its source. Automation of the specifications development process would also allow reduced design cycle times by eliminating much of the interpretation that must be done manually during the design process. Such a system would also allow an amount of back-annotation, since characteristics of a design could be traced back to their source in the requirements document.

1.2 ASPIN System

The work described here is part of a project called the Automated Specifications Interpreter (ASPIN) which is a tool being developed at Virginia Tech to improve productivity in creating specifications and requirements of digital systems. The ASPIN system accomplishes this by taking various forms of informal specifications and creating formal engineering models. From these formal models, simulation and further synthesis can be performed. The target formal model of the system is the VHSIC hardware description language (VHDL), which is fast becoming a standard language used in industry for modeling and synthesis.

Typically, specifications come in various forms of information. Block diagrams, timing diagrams, flow charts, and natural language are all commonly used to expressed the requirements of digital systems. Figure 1 shows an overview of the ASPIN system. The various forms of design information are entered into the system and mapped into the
conceptual graph knowledge representation. The natural language input is processed by the parser and the semantic analyzer in order to create the conceptual graphs. The parser creates trees of sentence structure that are used as input to the semantic analyzer. The semantic analyzer traverses the trees and builds up a conceptual graph that represents the information found in the natural language specifications. The resulting conceptual graphs are then processed and used to synthesize VHDL models simulating the behavior and characteristics of the system. The graphs are also used to create a graphical representation of the design information.

Figure 1: ASPI System Overview
Figure 2 gives more information about the organization of the system. Digital system requirements and specifications are entered by the specifications author. This information is processed and transformed into conceptual graphs. These conceptual graphs are then processed by the LINKER and GENERATOR. The GENERATOR creates a graphical representation that is fed back to the author for verification. The LINKER transforms the conceptual graph into a form that can be used by the Modeler's Assistant [12] to create the VHDL models.

![Diagram of ASPIN System Organization](diagram.png)

**Figure 2: ASPIN System Organization**

The Modeler's Assistant is an interactive graphical entry system for VHDL behavioral models of digital systems. The models are created by entering *process model graphs*
using the program's graphical interface. Process model graphs represent the structure of VHDL models. The graphs consist of nodes, which represent the VHDL processes, and arcs. The arcs represent the signals used to communicate between the processes.

Once the VHDL models are constructed using the Modeler's Assistant, the designer can continue system development by performing behavioral simulations for further verification. The VHDL models can also be used to perform further synthesis into low-level system models or hardware.

1.3 Research Approach

The approach of this work is based on compositional semantics of English sentences describing digital systems. A previously developed parser applies grammar rules to an input sentence to create a tree representing the syntactic structure of the English description. The parse tree serves as input to a semantic interpreter that creates conceptual graphs representing the meaning of the sentence. The conceptual graphs are built up by joining canonical conceptual graphs that represent words found in the input sentence. The parse tree directs the semantic analysis, which relies on a set of rules based on the grammatical structures found in the input sentence. The parser used in this work is written in C and the semantic analyzer is written in Quintus Prolog\(^1\). both pieces of software currently run on a Sun SPARCstation\(^2\) platform.

\(^1\) Quintus is a trademark of the Quintus Corporation

\(^2\) SPARCstation is a trademark of Sun Microsystems, Inc.
1.4. Thesis Organization

This thesis is composed of seven chapters. Chapter 1 contains an introduction to the goals and motivations for this work. Chapter 2 contains an introduction to natural language terminology and pertinent linguistic theory. Also included in Chapter 2 is a description of some related natural language processing systems. Chapter 3 describes the conceptual graph formalism, which is the target of the work described here. Chapter 4 provides a description of the parser used to create the trees of sentence structure necessary for the semantic analysis of the input sentences. Chapter 5 describes the organization of the semantic analyzer written in Prolog. The chapter includes a discussion of the ontology developed for the domain of digital system specifications. Chapter 5 also includes a description of the methods used to perform the semantic analysis. Various grammatical structures are presented along with a description of the strategy used to extract semantic information from them. Chapter 6 contains examples of conceptual graphs that are generated from descriptions of digital systems. Chapter 7 discusses the limitations of the semantic analyzer and suggests some directions for future work. Chapter 8 contains instructions on how to use the semantic analyzer.

There are four appendices included with this report. Appendix A contains a list of the vocabulary currently used by the parser and semantic analyzer. Appendix B contains a list of the grammar rules used by the parser to create the trees of sentence structure. Also contained in Appendix B is a list of definitions for all the grammatical abbreviations used in this work. Appendix C contains the complete Prolog source code for the semantic analyzer. Appendix D contains the list of sentences used in the test suite that exercise all of the semantic rules of the semantic analyzer. Finally, Appendix E contains a catalog of
the conceptual relations defined for the domain of natural language descriptions of digital systems.
Chapter 2. Background

2.1. Review of English Language Terminology

Since digital system descriptions do not use the full features of the English language, a technical sublanguage can be defined [3]. The grammar of the sublanguage is introduced here.

The sentence is a basic element of language and communication [2]. It is made up of ideas that are represented by various words and combinations of words. The words make up the parts of speech used in the English language. The parts of speech used in the technical sublanguage of this work are nouns, pronouns, verbs, adjectives, adverbs, prepositions, and conjunctions.

A noun can be a device or value, such as processor, instruction, and signal. Nouns can also be formed using verbs. A verbal noun is a verb in noun form. Such structures are known as deverbal nouns. Execution, executing, and to execute are all forms of the verb execute that can be used as nouns. Execution contains the noun forming suffix -tion. Executing is an example of a gerund, which is the present participle form of a verb used
as a noun. Finally, to *execute* is an example of the infinitive form of a verb, which can be used as a noun. Any word, phrase, or clause used as a noun is called a nominal.

Verbs can be used to express actions, events, or states of being. Verbs are the most important part of sentences since they express the primary relationship between words in a sentence. Most of the meaning derived from a sentence is taken from the verb and its use. Verbs can be used in one of two types of voice. The voice of the verb indicates whether the subject of the sentence performs the action, or is the object of the action expressed by the verb. In active voice, the noun phrase in the subject of the sentence performs the action. For example, the sentence *the register latches the data, and increments it* is in active voice. Therefore, *the register* performs the latching and incrementing activities. This sentence can also be expressed in passive voice to convey the same information. The sentence *the data is latched and incremented by the register* is in passive voice, indicating that *the data*, the subject of the sentence, is the object of the latching and increment actions.

Stative verbs are a form of the verb *be*. An example of the use of a stative verb can be seen in *the computer is a 32-bit machine*. Stative verb structures, also known as attributive predicates, come in several different forms. The example given above has a nominal as the complement of the stative verb. An adjective can also be used as a complement of a stative verb as in *the computer is fast*. Also, a prepositional phrase can serve as a complement as in *the data is in the register*.

Adjectives and adverbs are words that are used to modify other parts of speech. Adjectives modify nouns and pronouns. Adverbs modify verbs, adjectives, and other
adverbs. In the sentence the 8-bit data is latched asynchronously, 8-bit is an adjective used to modify the noun data. Asynchronously is an adverb that is used to modify the verb latched. Adjectives can take the form of ordinals such as first, second, and last. Adjectives can also be cardinals, which are numbers. An example of an ordinal and cardinal used as adjectives can be seen in the sentence the last byte of data is added to the two bytes which have already been shifted in.

Prepositions are used to express the relationship between two noun phrases or a noun phrase and a verb. Examples of prepositions can be found in the following phrases: on the bus, with one parity bit, and in the register. Prepositions provide important semantic information by indicating the type of role or relationship between the object of the prepositional phrase and the noun phrase or verb it modifies.

Conjunctions function as connectors of words, phrases, or clauses. There are two types of conjunctions, coordinatiag conjunctions and subordinating conjunctions. Coordinating conjunctions, such as and, but, and or, connect sentence elements of equal grammatical rank, i.e. nouns with nouns, independent clauses with independent clause, etc. Subordinating conjunctions, such as when, if, and because, connect dependent clauses with main clauses. A clause is a group of words containing a subject and a predicate. The system described here supports the used of various conjoined structures. The methods of processing conjunctions vary according to the type of structure present in the input sentence.

Sentences can be classified by their structure. The structure is determined by the organization of clauses. Sentences can be simple, compound, complex, and compound-
complex. A simple sentence is formed by a single independent clause. A compound sentence consists of two or more independent clauses joined with coordinating conjunctions. A complex sentence is formed by an independent clause joined with one or more dependent clauses. A compound-complex sentence consists of two or more independent clauses and one or more dependent clauses. Examples of the types of sentence structure are shown in Table 1. The English language parser and semantic analyzer described here supports all of these types of sentence structures.

**Table 1: Types of English Sentences**

<table>
<thead>
<tr>
<th>Type of Sentence</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>Simple Sentence</td>
<td>The register increments the data.</td>
</tr>
<tr>
<td>Compound Sentence</td>
<td>The register increments the data and the processor fetches the next instruction.</td>
</tr>
<tr>
<td>Complex Sentence</td>
<td>The register increments the data while the processor fetches the next instruction.</td>
</tr>
<tr>
<td>Compound-Complex Sentence</td>
<td>After the mode is selected, the register increments the data and the processor fetches the next instruction.</td>
</tr>
</tbody>
</table>

2.2. Introduction to Linguistics

A major part of modern formal language theory comes from the work done by Noam Chomsky in the 1950s. In 1957 Chomsky published the book *Syntactic Structures* which sought to describe the syntax of natural language using simple replacement rules and transformations. Chomsky stated that human language is infinite, but because of a person's understanding of the structure of language, he can create and comprehend sentences that are new to his experience. He defined a phrase structured grammar in
which elements of the sentences are identified in terms of their parts of speech. A phrase structure grammar starts at the sentence level and defines its parts. The parts are then broken into sub-parts. The redefining continues until the terminal, or words of the sentence, are finally reached. They are of the form:

\[ X \rightarrow Y \; Z \]

and are read "an X can be formed by a Y followed by a Z." For example, some simplified rules from the ASPIN grammar are shown below.

\[
\begin{align*}
S & \rightarrow SS . \\
SS & \rightarrow N \; PRED \\
N & \rightarrow NP \\
NP & \rightarrow DET \; NOUN \\
PRED & \rightarrow AVS \; N \\
AVS & \rightarrow VERB \\
DET & \rightarrow the \\
NOUN & \rightarrow signal, \; register \\
VERB & \rightarrow resets
\end{align*}
\]

(where S is a sentence, SS is a simple sentence, N is a nominal, PRED is a predicate, NP is a noun phrase, DET is a determiner, VERB is a verb, and NOUN is a noun).

The first rule reads "a sentence can be formed by a simple sentence followed by a period." Similarly, the second sentence is read "a simple sentence can be formed by a nominal followed by a predicate."

By replacing non-terminals in rules with strings of terminals (words), we can construct a sentence S.

\[
\begin{align*}
AVS & \rightarrow \text{resets} \\
NP & \rightarrow \text{the register} \\
N & \rightarrow \text{the register} \\
PRED & \rightarrow \text{resets the register}
\end{align*}
\]
NP -> the signal
N -> the signal
SS -> the signal resets the register
S -> the signal resets the register.

The derivation of the sentence can be represented in a "phrase marker," which is a tree of the sentence structure. The phrase marker for the sentence is shown in Figure 3.

Figure 3: Grammatical Structure Tree for the register is reset.

The most complex grammar that Chomsky developed was the transformational generative grammar. This type of grammar was made up of phrase structure rules, transformational rules and morphophonemic rules. Transformational rules perform operations such as changing a sentence from active voice to passive voice. The sentence
the signal resets the register is in active voice. Application of a transformation rule could change the sentence to the register is reset by the signal, which is in passive voice. Note that the subject and verb do not agree. One of the purposes of the morphophonemic rules is to correct agreement problems between the subject and verb of a sentence.

Chomsky's early theories were generally accepted and triggered a great amount of research in linguistics, which later pointed out some of the theory's shortcomings. Chomsky's theories relied solely on syntax and ignored semantics. Such an approach allows for the construction of sentences that, although syntactically correct, are semantically anomalous. Another difficulty arises when a sentence structure can have two meanings. The chickens are ready to eat is an example given in [9]. This sentence can be interpreted two ways: either the chickens are hungry and ready to eat their food, or the chickens have been cooked and are ready for human consumption.

In 1967 Charles J. Fillmore presented a paper called A Case for Case at a conference about linguistic theory. In this work, Fillmore attempted to deal with some of the problems of traditional linguistic theory by presenting a modified form of case. Traditional case forms show the relationship of each word to the other words in a sentence. The English language has three cases: nominative, possessive, and objective. The nominative case is used for words that are the subject of the verb. Also, the predicate nominative is used for a noun that is related to the subject of a stative verb as in the accumulator is a register. In this sentence, accumulator is the nominative case and register is the predicate nominative case. The possessive case shows possession and can be seen in the sentence the CPU's memory is cleared. The objective case is used for nouns and pronouns that are the objects of verbs. Fillmore made the point that the
traditional case forms dealt only with the surface structure of a sentence and had no significance to the meaning of a sentence. For example, when a sentence is changed from active voice to passive voice, the cases of the words in the sentence change even though the sentence expresses the same information. In the sentence the signal resets the register, signal is in the nominative case and register is in the objective case. When the sentence is transformed into passive voice, as in the register is reset by the signal, register is now nominative and signal is now objective. This problem prompted Fillmore to introduce a modified set of case notations that identify the roles that words play with respect to verbs described in sentences. Some of the cases that Fillmore introduced are shown below in Table 2.

<table>
<thead>
<tr>
<th>Case</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>AGENTIVE</td>
<td>the case of the entity that is the instigator of the action identified by the verb</td>
</tr>
<tr>
<td>INSTRUMENTAL</td>
<td>the case of the entity that is used or causally involved in the action or state identified by the verb</td>
</tr>
<tr>
<td>LOCATIVE</td>
<td>the case which identifies the location or spatial orientation of the state or action identified by the verb</td>
</tr>
<tr>
<td>OBJECTIVE</td>
<td>the case of the entity that is affected by the action or state identified by the verb</td>
</tr>
<tr>
<td>DATIVE</td>
<td>the case of the animate being affected by the state or action identified by the verb</td>
</tr>
<tr>
<td>FATICIVE</td>
<td>the case of the object or being resulting from the action or state identified by the verb</td>
</tr>
</tbody>
</table>
Fillmore also introduced case frames, which are a method for identifying the specific cases allowed for any particular verb. The case frame of a verb indicates the relationships required in a sentence containing that verb. The case frame also indicates relationships that are optional in a sentence containing the verb. For example, a case frame for the verb connect contains three cases: agentive, instrumental, and locative. In the sentence the bus connects the processor to the memory, bus is in agentive case, processor is in objective case, and memory is in locative case.

The case grammar introduced by Fillmore provides a basis for several of the elements of the ASPIN semantic analyzer. Many of the semantic roles defined in this work are similar to those developed by Fillmore. Also, the use of canonical graphs, described in Chapter 5, is an extension of the case frames described by Fillmore.

2.3. Related Research

Prior work in the area of modeling from natural language specifications has generally focused on software systems rather than hardware design. One notable exception is John Granacki's work with the PHRAN-SPAN natural language interface for the specifications of digital systems. Following are reviews of Granacki's work as well as two other projects dealing with the transformation of natural language into conceptual graphs.

2.3.1. PHRAN-SPAN

John Granacki initiated work to develop a natural language interface known as PHRAN-SPAN for specifications of digital systems [4,5]. The interface was developed for
ADAM, the USC Advanced Design Automation System [6]. The target language for the system is the USC Design Data Structure, which was designed to support and facilitate the synthesis of digital hardware systems [7]. The Design Data Structure supports the representation of data flow, timing and sequencing information. Although Granacki's work is concentrated on behavioral descriptions of digital systems, the Design Data Structure can also be used to represent structural information as well as physical information. The system uses a parser called PHRAN (PHRasal ANalyzer) to construct an internal representation of the information contained in a specification sentence. PHRAN detects patterns in the sentences and uses a database of pattern-concept pairs to map those patterns to a small set of concepts that are based on conceptual dependencies [2]. In conceptual dependency theory, knowledge is represented by the interrelationship of a small set of primitive acts. The sentence patterns detected by PHRAN are templates of phrasal constructs that can consist of a word, literal string, phrase or sentence. Associated with every pattern is a concept. PHRAN also makes use of some heuristics to identify ambiguity that often exists in nominal compounds used in the English language. Attempts to resolve the ambiguity are made using a set of rules designed to identify the correct usage of the words contained in the nominal compound.

The concepts in Granacki's work represent the primitive acts associated with conceptual dependencies and are expressed in an intermediate format. There are less than 20 concepts grouped into classes that deal with such things as information transfer, causal and temporal relationships, and declarations.

The semantic model of the natural language specifications is mapped to a model of digital system behavior by a program called SPAN (SPrécification ANalysis). SPAN's
role in the PHRAN-SPAN system is similar to that of the LINKER and GENERATOR of the ASPIN system described in Chapter 1. SPAN identifies the major structures for the behavioral model. SPAN also identifies the missing elements of the model. SPAN integrates the information that is contained in multiple sentence descriptions of system behavior and detects ambiguity that occurs. This part of the system is also able to identify the source of the ambiguity and report it to the user.

The work described here has many similarities as well as differences to the PHRAN-SPAN system described above. Both systems accept informal descriptions of digital systems in an effort to model the information contained in the English sentences. Also, both systems target a formal design notation in order to facilitate synthesis.

The PHRAN-SPAN system is based on a small set of primitive concepts that use conceptual dependency theory to represent the information contained in the behavioral descriptions of digital systems. This work, however, utilizes a large set of concepts which are organized into a hierarchy of types based on generalization and specialization. The hierarchy is several levels deep and allows for precise classification of digital system concepts. A large set of concepts allows more versatility and specificity in the interpretation of the system descriptions than a system based representations described by the interrelationship of a small set of primitive concepts. Also, the PHRAN-SPAN system deals primarily with the specification of system behavior. ASPIN deals not only with behavioral descriptions, but with structural descriptions as well.
2.3.2. Sowa and Way's PLNLP System

The work described in this thesis uses an algorithm similar to one developed by John Sowa and Eileen Way [8]. They began work on a semantic interpreter that utilized conceptual graphs to represent the meaning contained in natural language sentences. The system uses a syntactic parser to create trees that show the syntactic structure of a sentence. A semantic interpreter then takes the parse tree and generates a conceptual graph representing the meaning of the sentence. The parser used in the project is written in the Programming Language for Natural Language Processing (PLNLP), which is an augmented phrase structure language implemented in LISP/370 [15]. The parser uses a dictionary that contains over 70,000 words and a set of syntactic rules that is capable of handling almost any English language sentence. The technique of fitted parsing is also used to handle sentences that are grammatically incorrect. Fitted parsing occurs when an attempt to parse a sentence fails to produce a tree covering the whole input sentence. The resulting segments of sentence are analyzed and pieced together to form the most reasonable parse of the sentence. Because the parser is a strict syntactic parser, the dictionary can be simple. The parser can also handle sentences that conceptual parsers or detailed syntactic parsers might misinterpret since it uses fitted parsing.

The semantic interpreter for this system is also written in PLNLP. The basic data structure of PLNLP is the record, so each node in the conceptual graph is represented by a record. The records contain named attributes with associated values. These values can be atoms or pointers to other records representing other nodes in the graph. The semantic analyzer used by this system makes use of a library of canonical graphs that define the constraints on the concepts and relations that may be attached to concepts of a
given type. The four basic operations of conceptual graphs (copy, restrict, join, and simplify) are implemented as PLNLP subroutines. The interpreter starts analysis with a parse tree from the PLNLP parser. The parse tree determines the order that the canonical graphs of the input words are to be joined.

The system also handles two cases of ambiguity. In cases where words have more than one canonical graph (lexical ambiguity), the system maintains a set of multiple candidate graphs through the analysis. At the end of processing, all the complete graphs are examined and the best one is chosen. Determination of the best graph is accomplished by counting the number of concepts. The graph with the fewest concept nodes is preferred since a lower number of concepts indicates a greatest number of joins were made on that graph. Typically, the greater the number of joins, the less ambiguous the graph. Structural ambiguity occurs when the point of attachment for a subtree is not uniquely determined by the grammar. Sowa gives the example of John went to the chair by the window. In this sentence, the phrase by the window is in a structural position that usually modifies the verb, but here the phrase modifies chair. To handle such ambiguities, the system employs a technique of node moving. When the interpreter encounters such an ambiguity, the phrase node is moved and attempted to be attached to another concept in the sentence.

The PLNLP system developed by Sowa and Way targets the general English language. Their work is done to simply model the knowledge found in English sentences. The work described here deals specifically with the interpretation of digital system specifications. By narrowing the scope the input language, the analysis can be more precise and also more efficient. The semantic analysis reported in this thesis is done in
an effort to facilitate synthesis of formal design models. This applies strict constraints on the types of relations that can be defined and on the classification of the type hierarchy.

The method of analysis used by this system is very similar to the method used by Sowa and Way. Both systems can resolve lexical ambiguities, but the approaches are different. Sow and Way's system constructs multiple graphs that represent different interpretations of the ambiguity, and chooses the best graph at the end of processing. This system uses a set of role markers which use semantic and syntactic information to create a single graph representing the best interpretation of the ambiguity.

2.3.3. Pazienza and Velardi's Natural Language Understanding System

Maria Teresa Pazienza and Paola Velardi developed a system for Italian text understanding and intelligent information retrieval at the IBM Rome Scientific Center. The system was designed to analyze press agency releases and provided an automated way to provide answers to queries made about the information. The input to the system is a database of sentences in the domain of economics and finance. The input sentences are parsed by a syntactic parser with grammar rules containing a list of the first terminal symbols reachable from a given production. The parse trees serve as input to a semantic analyzer, which makes use of a conceptual dictionary. The dictionary contains semantic information as well as word usage information, and is implemented using conceptual graph theory. A set of about 50 conceptual relations were defined to express the relations found in the domain of language under consideration. The relations were grouped into three classes: role, complement, and link. The first class defines a set of relations that are used to relate the role that a concept plays with respect to an action, function, or event.
Example of role relations include AGNT, MEANS, and CAUSE. Complement relations link entities to concepts that describe their structure. They also link actions to concepts that describe the circumstances of those actions. In the phrase *to be late*, the relation TIME could be used to relate the concept *late* to the verb *to be*. The third class of relations, called *link* relations, are used for concepts in the same context (such as *driver* and *car*, or *banker* and *money*). Link concepts describe how concepts compare with respect to some role. For example, for the phrase *the book in the library*, INCLUSION is a link relation that could be used to relate *book* and *library*.

The semantic knowledge base used by the system is based on pragmatics (word usage, contexts, and figures of speech). It is difficult to classify concepts in a pragmatic approach since words that may be similar in one type of usage may be very different in another. For example, the use of the word *mouse* is very different in the following two sentences: *the mouse eats the cheese* and *the cat eats the mouse*. The verb for both sentences is *eat*, which requires a FOOD type to fill the role of object and an ANIMATE to fill the role of agent. *Mouse* is classified as a subtype of ANIMATE and *cheese* a subtype of FOOD. Therefore, there is no problem with the first sentence. In the second sentence, however, *mouse* is not a subtype of FOOD, but a subtype of ANIMATE. But, all ANIMATEs can be thought of as possibly filling the role of FOOD for some other ANIMATE. Thus, *mouse* has two very different uses.

To deal with this problem, two kinds of types are defined. One, called natural types, classify concepts based on the type of entities and actions. The second, called role types, describe the function or modifier of some natural type. Therefore, for the example given above, a role type can be defined declaring that food is a role that an ANIMATE type can
Having a role type and natural type hierarchies aids in the analysis of inference that is common in natural language. Separate hierarchies also prevents the need for an overly complicated single hierarchy lattice.

Pazienza and Velardi's system uses a set of syntax-to-semantics rules that convert syntactic structures to semantic roles. For example, for the structure of a noun phrase X followed by a prepositional phrase Y containing the preposition for, conversion rules can be defined describing three different roles.

\[
\begin{align*}
\text{Y RECIPIENT of X} & \quad \text{(gift for the student)} \\
\text{Y CAUSE of X} & \quad \text{(delay for traffic jam)} \\
\text{Y PURPOSE of X} & \quad \text{(agreement for funding)}
\end{align*}
\]

Examples of these types of structures are shown in parentheses. The determination of the correct roles is done using the natural type and role type hierarchies.

The system developed by Pazienza and Velardi is similar to the work described here. Both systems perform semantic analysis in an effort to create conceptual graph representations of knowledge contained in natural language statements. The conceptual graphs representations in Pazienza and Velardi's system are used to answer queries about the knowledge existing in the system, while the system described here uses the conceptual graph representation to synthesize formal models. The input to their system is Italian text in the domain of economics and finance. The domain studied here is that of digital system specification.
Pazienza and Velardi's use of syntax-to-semantic rules requires a large set of rules for every grammatical structure encountered in their sublanguage. Their method of analysis also requires very extensive role type and natural type hierarchies. The work described here has a small set of semantic rules based on syntactic structures. The semantic information is centralized since information is contained in canonical conceptual graphs and a single type hierarchy. The type hierarchy defined in this work corresponds to Pazienza and Velardi's natural type hierarchy. Also, the ASPIN semantic analyzer uses a database of knowledge that is used for inference, which is another function of Pazienza and Velardi's role types.
Chapter 3. Conceptual Graphs

3.1. Introduction

The target representation for this work is the system of conceptual graphs developed by John Sowa [11]. Conceptual graphs are a type of semantic network often used in artificial intelligence systems. The earliest forms were developed by Charles Peirce in the nineteenth century and were called existential graphs. Peirce developed existential graphs to serve as a graphical notation for symbolic logic. Conceptual graphs are an ideal target representation for semantic analysis since they represent mental models of information. Conceptual graphs are used for this application since they map directly into formal logic, which can be used for reasoning and consistency checking.

By definition, a conceptual graph is a finite, connected, bipartite graph. The graphs are finite since any graph representing some knowledge can have only a finite number of concepts and conceptual relations. Any graph that is not connected is simply a group of individual conceptual graphs. Conceptual graphs consist of two types of nodes. The first type of node is the concept, which can represent an object, action, or feature. The other type of node is the conceptual relation, which identifies the role of each conceptual
entity. Every arc in a conceptual graph links a concept to a conceptual relation. Therefore, conceptual graphs are bipartite.

![Figure 4: Graphical Representation of a Conceptual Graph](image)

Conceptual graphs can be expressed graphically as well as in linear text. In graphical representations, the concepts are drawn as boxes and the conceptual relations are drawn as ellipses. The arcs linking the concepts and relations are drawn as arrows. Figure 4 shows the graphical representation for a conceptual graph. For this application, the concept nodes contain two labels. These labels specify the conceptual type and the referent of the concept. The conceptual relation nodes contain two labels which specify the role and the referent of the relation.

The referent field can contain several different types of markers. An asterisk '*' in the referent field of a concept is called a *generic* marker and represents an unspecified individual of the given type. An *individual* marker is represented by a pound sign '#' followed by some integer identification number. The individual marker is used to identify a particular instance of a concept type. The referent field can also take the form of a disjunctive set. This form can often be seen in the referent field of conceptual relations contained in canonical conceptual graphs. An example can be seen in the canonical graph of *load* shown in Figure 6. The conceptual relation *destination* contains
the set of referents \{ in, into \}. In this case, the disjunctive set referent indicates that the destination relation can be indicated by the preposition in or into.

In the linear text representation of conceptual graphs, the concepts are contained in square brackets and the relations in parentheses. Arrows are used to represent the arcs in the graph. An extra label is used in this application of the linear form to identify the concepts using a concept number. Therefore, the linear text representation of the conceptual graph form shown in Figure 4 is as follows:

\[
\begin{align*}
[1 & : \text{TYPE : referent}] \\
  & -> (\text{relation : referent}) -> [2] \\
[2 & : \text{TYPE : referent}]
\end{align*}
\]

This graph is read "the relation of concept 1 is concept 2." For example, the graph below is read "the location of the value is memory."

\[
[\text{VALUE : value}] -> (\text{location}) -> [\text{MEMORY : memory}]
\]

Conceptual graphs model knowledge by classifying concepts and defining the relationships between concepts. Figure 5 gives an example of a sentence and a conceptual graph that is used to model the information contained in the statement. In this example, there are four concepts and three conceptual relations linking them. The processor is a subtype of DEVICE and fills the role of the agent of the read action. Similarly, data is a subtype of VALUE and is the object of the read action. Memory, which is also a subtype of DEVICE, acts as the source, or location where the data is read from.
Figure 5: Conceptual Graph for "The processor reads the data from memory"

This graph is represented in the linear form as follows:

```
[ 1: READ : read ]
  - > ( agent )    - > [ 2 ]
  - > ( obj )      - > [ 3 ]
  - > ( source : from ) - > [ 4 ]
[ 2 : PROCESSOR : processor ]
[ 3 : DATA : data ]
[ 4 : MEMORY : memory ]
```

The concepts represented by the boxes in the above graph denote instances of the concept type specified, rather than any general concept of that type. In database systems, unique markers called surrogates are used to identify particular individuals. In conceptual graphs, surrogates are represented by individual markers, which are found in the referent field of the graph of a concept. The referent label is used to distinguish individual instances from types. For example, in the graph given above, [ 2 : PROCESSOR : processor ] indicates an instance of a type of PROCESSOR. The word processor is used as the referent for the instance of PROCESSOR since it is the most specific marker that can be derived from the sentence. When no individual marker exists, an asterisk is used
in the referent field. When a concept contains an asterisk in its referent field, it is said to be a generic concept. The concept [ 2 : PROCESSOR : * ] would refer to an unspecified instance of a PROCESSOR.

3.2. Semantic Model

Without an understanding of the relationships and context of entities within the domain of language being studied, conceptual graphs have no meaning. The set of relationships that concepts have to other concepts, as well as the background knowledge needed to understand these relationships is contained in the system's semantic model. The main parts of the semantic model for this work are the set of canonical conceptual graphs, the conceptual type hierarchy, and the set of conceptual relations linking the concepts.

Every concept within the technical sublanguage developed here has a canonical conceptual graph that specifies the constraints by which attachments can be made with other conceptual graphs. Canonical graphs also define the type of the given concept. They serve as templates by which other conceptual graphs can be generated. The canonical graphs consist of a set of relations by which other canonical graphs can be attached. With the relations a "slot" is defined that designates the valid concept types that are allowed to be attached to the canonical graph using that relation. When an attachment is attempted using a given relation, the attaching conceptual graph must be a subtype of the required type defined by the slot of the canonical graph. Canonical graphs are discussed in more detail in Chapter 5, section 3.
Sowa defines a set of formation rules that may be used to derive new conceptual graphs from other conceptual graphs. The four formation rules are copy, restrict, join, and simplify. Copy states that an exact copy of a canonical graph is also canonical.

Restrict replaces the type label of a concept with the label of a subtype. The changes are only allowed if the referent conforms to the subtype. For example, use of the restrict rule can convert the concept [ PROCESSOR : M68HC11 ] to [ MICROCONTROLLER : M68HC11 ]. The conversion is permitted since M68HC11 is an instance of PROCESSOR as well as MICROCONTROLLER.

The restrict rule can also be used to convert a generic concept to an individual concept. For example, the concept [ INSTRUCTION ] can be restricted to [ INSTRUCTION : branch_instruction ].

The join formation rule combines two graphs by merging identical concepts. In order for two concepts to be merged, their types and referents must be the same. If one of the concepts is generic, it can be restricted to match the concept it is being merged with. Once the concepts are merged, all of the relations attached to the two concepts are attached to the merged concept. Figure 6 shows the canonical graphs for load and microcontroller.
Figure 6: Canonical Graphs of "load" and "microcontroller"

These two graphs can be joined using the relation agent by restricting the graph of load as shown in Figure 7.

Figure 7: Canonical Graph of "load" After Restriction

In Figure 7, the generic concept PROCESSOR has been restricted to MICROCONTROLLER. Figure 8 shows the conceptual graph resulting from the join of the canonical graphs in Figure 6.
Figure 8: Join of Canonical Graphs in Figure 6

When canonical graphs are joined, sometimes redundant relations exist. The formation rule simplify deletes any duplicate conceptual relations in a graph. Relations are considered duplicates when they are linked to the same concepts in the same order. Since the duplicate relations contain redundant information, the simplify formation rule does not cause a loss of information in the resulting conceptual graph.

The second main part of the semantic model defined for this work is the conceptual type hierarchy which is a partial ordering defined over the set of types based on levels of generality. The structure of the conceptual type hierarchy is shown in Figure 9. The hierarchy is a lattice, therefore any two conceptual types classified in the hierarchy have a common supertype as well as a common subtype. At one end of the lattice is the universal type, which is the supertype of all other types. At the other end is the absurd type, which is a subtype of all other types. Each node in the lattice represents a concept type within the domain of digital system specifications. The concept type hierarchy is discussed in more detail in Chapter 5, section 2.
Figure 9: Structure of the Conceptual Type Hierarchy

The final part of the semantic model essential for semantic analysis is the set of conceptual relations. The conceptual relations describe the way concepts are assembled by specifying the roles that the concepts play with respect to one another. The set of conceptual relations is typically small since the concepts represent most of the knowledge in a conceptual graph. The set of relations used in this work is presented in Appendix E. The appendix also contains formal definitions for the relations.
Chapter 4. Syntactic Processing

4.1. Introduction

Syntactic processing is the step in which an input sentence is converted into a hierarchical structure that corresponds to the units of meaning in a sentence [10]. The process is generally referred to as parsing. Parsing is important to natural language systems because it defines constituents which the semantic processing system can use. Otherwise, the semantic processing system must define its own constituents, which is difficult and computationally expensive. Without the use of grammatical facts, accurate interpretation of sentences becomes difficult and sometimes impossible.

There are three elements commonly used in syntactic analysis systems. One is a grammar, which is a representation of the structures used in a language. The second element is a dictionary, which defines the possible syntactic uses for words in the language. The final element is a parser, which is a procedure used to compare the grammar of a language to input sentences to produce parsed structures. The most common method of representing grammars is through the use of production rules as introduced by Noam Chomsky in his book *Syntactic Structures*. Figure 10 lists some of
the grammar rules used by the ASPIN parser. The notation for these rules was discussed in Chapter 2, section 2.

```
S  ->  SS .
SS  ->  N PRED
N  ->  NP
NP  ->  DET NOUN
PRED  ->  AVS N
AVS  ->  VERB
DET  ->  the
NOUN-> signal, register
VERB  ->  resets
```

**Figure 10: Example Grammar Rules Used by the ASPIN Parser**

The constituents of these production rules are made up of terminal and non-terminal symbols. Non-terminal symbols can be broken down into smaller constituents using other production rules. Terminal symbols correspond to the actual words used in the English sentence. For example, in Figure 10, SS (simple sentence), is a non-terminal symbol that can be broken down into an N (nominal) followed by a PRED (predicate). Terminal symbols are words like *register, signals,* and *reset* that cannot be further decomposed using a grammar rule and are embodied in the ASPiN dictionary.

In order to parse the sentences, the syntactic processor must identify the words in the sentence, and match these sequences of words with the grammar rules. A complete parse tree is found when a production rule for S is satisfied and spans from the first word of the sentence to the last word of the sentence. There are two ways to approach this analysis: top-down parsing and bottom-up parsing.
Top-down parsing begins with the rules that form a sentence S. The constituents from those rules are then examined and rules for those constituents are found. This type of analysis applies grammar rules in order to decompose the sentence from the highest structure level S down to the level of the terminal words in the sentence. If the path of sentence decomposition leads to a set of constituent symbols that can be represented by the words in the sentence, then a parse tree for the sentence is formed. If the words of the sentence cannot represent the series of symbols necessary, then a new path from the highest order symbol S is chosen. Without the use of good heuristics, this search requires much computational time due to the backtracking that must take place when words of the sentence do not match the chosen symbols.

The other method of parsing, bottom-up parsing, starts with each word of the sentence and works backwards. The grammatical category for each word is identified and are matched against the right-hand side of the production rules. If a match is found, then the sequence of rules can be combined into the constituent identified on the left-hand side of the rule. These larger constituents are then used to combine with each other and the other remaining words of the sentence to build even higher level constituents. Processing continues up the sentence structure tree until a sentence symbol S is able to be constructed that covers all the words in the input sentence.

Winograd described a method of parsing known as chart parsing in [14] that is more efficient than standard top-down and bottom-up parsing algorithms. The algorithm organizes the formed structures into a chart that can be used to form larger grammatical structures. The chart parser is more efficient that standard top-down and bottom-up parsers since the only structures that are built up are those relevant to the input sentence.
Also, since the structures are contained in a chart, they need only be formed once instead of multiple times as is the case in search and backtrack algorithms.

A chart can be thought of as a network of vertices representing points in the sentence. The vertices are linked together by arcs that represent the formation of a grammatical structure. The edges are given a label that identifies the name of the structure represented by the edge. The beginning and ending points of the formed structure is given by the vertices linked by the edge. Figure 11 shows an example of a simplified parse chart for the signal resets the register.

Figure 11: Simplified Parse Chart for "the signal resets the register"
An example using a chart parsing algorithm is given below in section 2 of this chapter.

4.2. ASPIN Parser

The parser used to create the trees of sentence structure is a bottom-up, chart parser that uses a phrase-structured grammar. The grammar was constructed manually through the study of various VLSI product descriptions [13]. It contains over 120 productions using 48 non-terminal symbols. The grammar is easily extendible to include specialized structures not recognized by the current rules. A complete listing of the grammar can be found in Appendix C.

The ASPIN parser makes use of a dictionary that specifies the lexical categories for over 1200 words. When a sentence word is scanned by the parser, the program searches the dictionary for its lexical categories. The lexical categories are then added to the chart used by the parser to build up grammatical structures.

Consider the parsing process for the sentence *the signal resets the register*. The parser reads in the first word, *the*, and searches its dictionary for its grammatical category. *The* is found to be a determiner, DET, and is entered into the parse chart. Since the parser builds up a chart of structures one word at a time, the chart starts with one entry, the word *the*. Figure 12 shows the shape of the parse chart after the word *the* is entered. The set of grammar rules is then examined for any production rule with DET as its right-most constituent. None exist, so the next word from the input sentence is examined.
The next word in the input sentence is *signal*, which is classified in the dictionary as a noun. After it is entered into the chart, the set of grammar rules is checked for a production rule that has NOUN as its right-most constituent. The first HEAD rule is found and an edge labeled HEAD is added to the chart since it has only one constituent, which is filled by the NOUN structure. Since no more rules exist that have NOUN as its right-most constituent, the same rule search is done for the HEAD structure that was produced by the NOUN using the first HEAD rule. This search finds the twelfth noun phrase rule, NP12. Rule NP12 states that a noun phrase can be formed by a determiner (DET) followed by a head (HEAD). Since the right-most constituent of this rule is filled by the HEAD structure, the parser then examines the structures immediately to the left of the HEAD. Since a DET is immediately to the left of the HEAD, the rule NP12 is satisfied and a noun phrase NP is formed that spans the first two words of the input sentence. Further searches are done using the HEAD structure formed, and later the NP structure formed. The resulting chart is shown in Figure 13.

The parser continues to build up the parse chart until the period at the end of the input sentence is input and parsed. A simplified parse chart for *the signal resets the register* is
shown in Figure 11 of the previous section. The parse tree for this sentence is shown in Figure 14.

**Figure 13:** Parse Chart for the Words "the signal"

**Figure 14:** Parse Tree of "the signal resets the register"
After parse chart is completed, the parser examines it for any sentence structures, S, that have been formed. If the S structures span the entire sentence, then a valid parse tree has been created. Otherwise, no valid parse trees exist for the input sentence.

Tree #1

Tree #2

Figure 15: Two Parse Trees for "the input signals reset"
Since English words often have multiple grammatical categories, the parse chart can get complex, resulting in several syntactically valid parse trees. When the sentence *the input signals reset* is parsed, three syntactically valid parse trees result. Figure 15 shows two of the parse trees. In Tree #1 of Figure 15, *the input signals* is the subject of the sentence and *reset* is the verb. In Tree #2 of Figure 15, *the input* is subject of the sentence, *signals* is the verb, and *reset* is the object of the verb. Although these parse trees are syntactically correct, only one will lead to the correct interpretation.

![Parse Chart](image)

**Figure 16: Parse Chart for "a constant executes the bus"**

Figure 16 shows the parse chart for *a constant executes the bus*. This sentence does not make sense, yet it has the same parse chart structure as *the register is reset* shown in
Figure 11. This is an example of a sentence that is syntactically correct, but semantically nonsensical. It is the job of the semantic analyzer to reject such semantically nonsensical parse trees.
Chapter 5. Semantic Analysis

5.1. Introduction

The function of the semantic analysis is to take the sentence structure trees formed by the parser and create conceptual graphs representing the meaning of the sentences. The approach taken here is to represent the concepts within the sentences with canonical conceptual graphs which are stored in a database. A canonical graph for a concept is a conceptual graph that defines the conceptual relations by which other concepts may be attached. Canonical graphs also designate the type of concepts that the canonical graph may be linked with. Thus, canonical graphs define the constraints by which meaningful conceptual graphs are constructed.

The concepts in the technical sublanguage under consideration are organized into a conceptual type hierarchy. The hierarchy is based on levels of generality and provides information that is used in the attachment of graphs.
The semantic analysis of the ASPIN system is done by the Semantic Analyzer software. The parse trees generated by the parser described in Chapter 4 serve as input to the semantic analyzer. The analyzer is written in Prolog and uses several databases of information to perform the analysis. The processing portion of the semantic analyzer, called main, accesses the system databases. A database of semantic rules is defined for the grammatical structures used in the grammar of the ASPIN system. The rules guide the semantic analysis and construction of conceptual graphs. A database of canonical graphs is used by the Semantic Analyzer to set the constraints for attachments made within the semantic rules. Before a canonical graph is retrieved for a word in an input sentence, the system accesses a database of root words. The database consists of a set of Prolog facts that define the root words of all the words for which canonical graphs are defined. Figure 17 shows the organization of the Semantic Analyzer.

![Semantic Analyzer Organization Diagram](image-url)

Figure 17: Semantic Analyzer Organization
5.2. Concept Type Hierarchy

Digital system specifications written in English use a subset of the English language. Therefore, a technical sublanguage vocabulary and grammar can be defined. Since a subset of the English language is used, the full vocabulary of the English language is not supported.

A hierarchy of concept types defines the relationships between concepts of different levels of generality [11]. Each concept within the sublanguage is classified into the partial conceptual type hierarchy shown in Figure 18. The taxonomy is a lattice of six basic types generalizing several more specific subtypes. Table 3 contains definitions of the six most general types.

Table 3. Definitions of the Most General Conceptual Types

<table>
<thead>
<tr>
<th>Conceptual Type</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>OBJECT</td>
<td>Any unit of information, software or hardware.</td>
</tr>
<tr>
<td>ACTION</td>
<td>An activity or process, usually having a duration.</td>
</tr>
<tr>
<td>STATE</td>
<td>Stative verbs and arithmetic/logical relations.</td>
</tr>
<tr>
<td>STRUCTURE</td>
<td>An attribute describing the organization of a structured concept.</td>
</tr>
<tr>
<td>EVENT</td>
<td>A discontinuity in one or more actions.</td>
</tr>
<tr>
<td>MEASURE</td>
<td>An attribute of the dimension or quantity of an object or action.</td>
</tr>
</tbody>
</table>
Figure 18. Partial Conceptual Type Hierarchy

The hierarchy is arranged with the most general type, the universal type, as the root of the left side of the lattice. The more specific types are to the right. The root of the lattice on the right side is the absurd type. The universal type is a supertype of all other types. The absurd type is a subtype of all other types. Every concept in the domain of language under consideration is an instance of the universal type. Similarly, no concept is an instance of the absurd type.

The concept types defined in the lattice are like variables that represent an unspecified individual of that type. For example, when the word *microcontroller* is used in a
sentence, *microcontroller* represents the name of an individual whose type is MICROCONTROLLER. The particular microcontroller in question may have a serial number, which would be the identifier or individual marker that would distinctly represent that instance of a microcontroller. In general, English descriptions usually omit the individual marker and refer to the individual using a name (proper noun) or a type (common noun).

Consider the sentence *the M68HC11s contain 8k of read-only memory.* *M68HC11* is a subtype of MICROCONTROLLER in the hierarchy. No individual markers are given to specify which group of M68HC11s this sentence refers to. Also, since there is no marker, an M68HC11 in the group cannot be distinguished from another. Thus, the sentence refers to a group of M68HC11 microcontrollers by using the type name of the individuals.

<table>
<thead>
<tr>
<th>Concept Types</th>
<th>Subtype Examples</th>
</tr>
</thead>
<tbody>
<tr>
<td>DEVICE</td>
<td>circuit, unit, component, chip</td>
</tr>
<tr>
<td>PROCESSOR</td>
<td>computer, microprocessor, controller</td>
</tr>
<tr>
<td>LOGIC</td>
<td>adder, alu, gate, decoder</td>
</tr>
<tr>
<td>MEMORY</td>
<td>ram, flip-flop, eprom, latch</td>
</tr>
<tr>
<td>LOGIC_MEMORY</td>
<td>counter, timer, shift_register</td>
</tr>
<tr>
<td>CARRIER</td>
<td>bus, line, wire, pin</td>
</tr>
<tr>
<td>TRANSDUCER</td>
<td>sensor, detector, keyboard</td>
</tr>
<tr>
<td>COMMAND</td>
<td>software, program, instruction</td>
</tr>
<tr>
<td>ADDRESS</td>
<td>location, address, index, pointer</td>
</tr>
<tr>
<td>DATA</td>
<td>operand, sum, result, multiplier</td>
</tr>
<tr>
<td>CONSTANT</td>
<td>0, 1, low, high</td>
</tr>
<tr>
<td>INFORMATION</td>
<td>flag, signal, pulse</td>
</tr>
<tr>
<td>OPERATE</td>
<td>execute, add, complement, shift</td>
</tr>
<tr>
<td>MOVE</td>
<td>swap, transfer, exchange</td>
</tr>
<tr>
<td>READ</td>
<td>fetch, read, input, pop</td>
</tr>
<tr>
<td>WRITE</td>
<td>load, set, store, push</td>
</tr>
<tr>
<td>CONNECT</td>
<td>select, connect, address</td>
</tr>
<tr>
<td>PHYSICAL_STRUCTURE</td>
<td>pipeline, ring, contain, parallel</td>
</tr>
<tr>
<td>-------------------</td>
<td>----------------------------------</td>
</tr>
<tr>
<td>DATA_STRUCTURE</td>
<td>array, vector, string, stack</td>
</tr>
<tr>
<td>ACTION_STRUCTURE</td>
<td>sequence, concurrent, synchronous</td>
</tr>
<tr>
<td>AFFECT_EVENT</td>
<td>enable, disable, control</td>
</tr>
<tr>
<td>CALL_EVENT</td>
<td>call, return, interrupt, jump</td>
</tr>
<tr>
<td>CAUSAL_EVENT</td>
<td>initiate, activate, resume, invoke</td>
</tr>
<tr>
<td>TERMINATE_EVENT</td>
<td>stop, end, terminate, conclude</td>
</tr>
<tr>
<td>MEMORY_MEASURE</td>
<td>bit, byte, word, line</td>
</tr>
<tr>
<td>TIME_MEASURE</td>
<td>period, cycle, sequence, second</td>
</tr>
</tbody>
</table>

Since, in general, common nouns and verbs correspond to type labels, the task in developing the conceptual type hierarchy is determining where the nouns and verbs of the vocabulary fit into the lattice. The relationship between more specialized types and generalized types corresponds to the \textit{isa} relation used in inheritance systems. For example, any concept that is classified as a type of PROCESSOR is also a type of DEVICE and a type of OBJECT. Table 4 gives examples of subtypes from the conceptual type hierarchy.

5.3. Canonical Conceptual Graphs

A conceptual graph is made up of concept nodes and relation nodes. Every arc links a conceptual relation with a concept. Arbitrary combinations of concept nodes and relation nodes do not make sense. For example, the graph

\[
[ \text{ACTION} ] \rightarrow (\text{location}) \rightarrow [ \text{EVENT} ] \rightarrow (\text{condition}) \rightarrow [ \text{DEVICE} ]
\]

states that there is an ACTION whose location is an EVENT that occurs when there is a DEVICE condition. To rule out nonsensical sentences, a set of canonical graphs is defined that represents valid semantic situations.
Each word in the vocabulary of the sublanguage has a syntactic type and possibly a semantic type. The syntactic type is the lexical category of the word, and governs the grammatical use of the word, as discussed in Chapter 4. This information is used by the parser to build a tree of sentence structure. The semantic type of the word describes what roles or meaning the word can have in the interpretation of the English sentence. The semantic attributes of words are represented in canonical conceptual graphs. A canonical graph is a conceptual graph that represents the meaning of a word or grammatical structure. It also defines the relations by which joined concepts are attached. The canonical graphs accomplish this by providing slots in the form of generic concepts. These generic concepts are restricted when joined with another conceptual graph. These generic concepts constrain the kind of concept that may fill the slot, since in order for a join to take place, the attaching graph must be compatible with the generic concept.

```
[ LOAD : load ]
-> ( agnt )  -> [ PROCESSOR ]
-> ( obj )  -> [ VALUE ]
-> ( dest )  -> [ MEMORY ]
```

A canonical graph for the verb load is shown above. The graph contains three relations by which other concepts can be attached. In order for another graph to be attached using the agent relation, it must by a type of PROCESSOR. If the graph to be attached to this slot is not a type of PROCESSOR, then the joining of the two graphs will not occur. Similarly, for a graph to be attached to the canonical graph using the object relation, the attaching graph must be a type of VALUE.
Since English words can be used in several senses, it is often necessary for a word to have more than one canonical graph. A second canonical graph for the verb load is shown above. This canonical graph differs from the previous graph in the type of concept that can be attached in the agent role. The agent role in the canonical graph shown above can only be filled by a type of COMMAND, while the agent role in the previous canonical graph can only be filled by a type of PROCESSOR. This difference can be seen in the following two sentences.

The processor loads the data into the register.

The store instruction loads the data into the register.

5.4. Role Markers

Role markers are also included in the canonical graphs as referents of conceptual relations. They serve as an aid in the semantic analysis. Role markers direct the Semantic Analyzer during the attachment phase of analysis.

There are two types of role markers: literal and structural. Literal role markers correspond to actual words, such as prepositions and subordinate conjunctions, used in the input sentences. An example of a literal role marker can be seen in the destination conceptual relation of the canonical graph of load on page 52. In and into are markers
that direct the Semantic Analyzer to try to attach any adverbial prepositional phrases that begin with the words *in* or *into* using the conceptual relation destination.

Structural role markers tell the Semantic Analyzer how to link a word's syntactic use with its semantic use. For example, in the sentence

*The strobe rises.*

the syntactic use of the word *strobe* is different than the semantic use. Syntactically, *strobe* is the agent of the verb *rises*. Semantically, *strobe* is not the cause or agent of *rises*, but the object of that action. Therefore, the marker *agent* is included as a structural role marker in the object relation of the canonical graph for the verb *rises* as shown below. Structural role markers allow the attachment of graphs to be done semantically as well as syntactically.

\[
[ \text{RISE} : \text{rise} ] \rightarrow ( \text{obj} : \text{agnt} ) \rightarrow [ \text{VALUE} ]
\]

5.5. Attach Operation

The construction of the conceptual graphs centers around a set of rules that attach canonical graphs together. The attach operation consists of selection of the appropriate relation by which to join the two graphs under consideration, and the actual join itself. In order for the appropriate relation to be determined, certain information must be known about the two graphs. First, it must be known which of the two graphs is the head graph and which is the graph to be attached to the head graph. The type of the graph that is to
be attached to the head graph must also be known. Also, any role marker that pertains to the join must be known. Finally, associated with the head graph is a list of empty slots, which serve as points of attachment. These slots contain the type constraints and roles that are used to attach the two graphs. The canonical graph of load is given below, which shows an example of a set of empty slots. There are three points of attachment designated by the relations agent, object, and destination. For example, the role of destination can be filled by a subtype of DEVICE. For this role, a disjunctive set of role markers exists which indicates that the role marker in or into must be present in order for a graph to be attached to this slot.

\[
[ \text{LOAD} : \text{load} ] \\
\text{-> (agnt : by)} \quad \text{-> [PROCESSOR]} \\
\text{-> (obj)} \quad \text{-> [VALUE]} \\
\text{-> (dest : \{in | into\}) -> [DEVICE]} \\
\]

Given this information, the semantic analyzer uses a set of selection rules to determine the conceptual relation linking the two graphs. A flow chart for the attach operation is shown in Figure 19. The rules check the first empty slot of the head graph for the role marker constraints. If the role marker associated with the attachment of the two graphs in question does not match any of the required role markers, then the next empty slot of the head graph is tried. Otherwise, the type constraints for the empty slot in question are checked against the type of the graph which is to be attached to the slot. The attaching graph is the proper type of graph to fill the empty slot if its conceptual type is the same or a subtype of the required type designated by the empty slot. If this is verified, the
graph is attached to the empty slot using the conceptual relation defined by the slot and any marker that was associated with the attachment.

![Flow chart of the Attach Operation](image)

**Figure 19: Flow chart of the Attach Operation**

Failure to meet the type constraints prompts the semantic analyzer to check a set of inference facts. The inference facts provide information about various conceptual types that can be used to resolve inference common in natural language use. These facts are discussed in more detail in section 7 of this chapter.
If the inference facts cannot resolve the type mismatch of the attaching graph and the slot, then the next slot in the list of available slots is tried as an joining point for the attaching graph. Therefore, the selection process starts at the beginning using the next available slot in the canonical graph of the head concept. If attachment is not made after all the available slots have been tested, then a set of special selection rules is used. The first of these special selection rules check the role marker associated with the two graphs. If the marker is *if, when, or as a result*, the attaching graph is attached using the conceptual relation *condition*. If the marker associated with the two graphs is *before*, then the graph is attached using the conceptual relation *before*. It should be noted that these special rules do not use the list of available slots defined in the canonical graph of the head concept. Instead, they create a new slot and use the role marker associated with the attachment to determine the correct conceptual relation by which to join the two graphs.

If the graph still has not been attached at this point, then the supertype of the attaching graph is determined and the selection process starts from the beginning. The only change that will result from the use of the supertype is in the search for inference facts. Inference facts that apply to a conceptual type also apply to its subtype. The scan of inference facts, however, is done based on the concept type in question and does not check for facts relating to concept type's supertype. Therefore, use of the supertype is necessary to find all the inferencing information about a conceptual graph that is trying to be attached to another.

A detailed example of the processing of an input sentence can be found in Section 2 of Chapter 6.
5.6. Semantic Rules

The set of semantic rules used by the Semantic Analyzer is based on the grammatical structures used to parse the input sentences. Each node of the parse tree corresponds to a grammatical construct which has a corresponding Prolog rule in the Semantic Analyzer. Therefore, it is through a sequence of semantic rules that the input parse tree is traversed. Each node in the parse tree also corresponds to a subgraph of the conceptual graph constructed to represent the meaning of the input sentence. The semantic rules designate which graph of a set of sibling graphs is the main graph to which the other sibling graphs are to be attached. It is through this attachment of sibling graphs that the conceptual graph for the input sentence is built up.

Processing begins at the root of the parse tree and proceeds depth-first, along each subtree of each node. Path tracing continues until a leaf node is reached, at this point a graph is constructed. If the leaf node in question is a type of noun, verb, or adjective, a canonical graph is retrieved from the canon. Otherwise, an appropriate graph is created, as in the case of proper names and identifiers. The resulting graph is then passed up to its parent node. The graph continues to bubble back up the tree until an unprocessed sibling subtree is reached. The sibling subtree is then processed and the procedure is repeated. When all the subtrees of a given node have been processed, the graphs which have been passed up are then linked together. The semantic rules of the parent node dictate how the subtree graphs are attached and which of the subtree graphs is to be the main graph passed up for attachment at higher branches in the parse tree.
5.6.1. Adjective Rules

Semantic analysis is often difficult due to the richness of the English language. Adjectives are a particular problem since they modify nouns in many different ways. Adjectives often carry information that is important in modeling and synthesis. Therefore, when possible, they should not be attached to the conceptual graph using a general conceptual relation such as attribute, which might omit valuable semantic information.

Adjective structures can be formed not only by pure adjectives, but also by past participle and gerund forms of verbs. A past participle form of a verb used as an adjective can be seen in the loaded register is ready to be incremented. Past participle verbs usually modify a noun that is the object of the past participle verb. Therefore, the rule for this verb form retrieves the canonical graph for the past participle verb and attempts to attach the graph of the nominal which it modifies using the object relation. If the attachment succeeds, this new graph is added to the relative graph list where it can be joined to the main sentence graph in post-processing.

Present participle forms of verbs, such as loading, rising, and triggering, are often used as adjectives. They present more difficulty in analysis since nouns modified by these verb forms sometimes fill the role of object of the verb, and other times fill the role of agent. For example, in the rising signal triggers the latch, signal semantically fills the role of object of the verb rise. In the triggering signals load the latch on a 0-1 transition, signals fills the semantic role of agent of trigger. The semantic analyzer handles these type of structures by retrieving the canonical graph for the verb. The conceptual graph of
the modified noun is then attached to verb using the first slot containing an appropriate conceptual type.

Figure 20. Canonical Graph for the Adjective "8-bit"

Adjective structures consisting of pure adjectives are attached in a more descriptive way. Each adjective is given a canonical graph. The canonical graphs for adjectives identify the type of concepts they modify, and also the role by which they are attached. Since adjective canonical graphs define the roles they fill for other concepts, the arrows of the graphs point in the opposite direction of those for the other canonical graphs. Figure 20 shows two canonical graphs for the adjective 8-bit. These graphs indicate that 8-bit can be attached to a graph that is a subtype of DATA using the conceptual relation size and to a graph that is a subtype of DEVICE using the conceptual relation type.

5.6.2. Adverbial Rules

Adverbials, by definition, are words, phrases, or clauses that modify verbs, adjectives and other adverbs. Like adjectives, adverbials can also be difficult to interpret correctly. Adverbials can be formed from prepositional phrases, subordinate clauses, as well as simple adverbs. The semantic rules contain predicates that will handle five different
adverbial structures. Adverbials that are formed from pure adverbs are handled in much the same way as adjectives. Each adverb is given a canonical graph. The canonical graphs for the adverbs identify the type of concepts they modify, and also the role by which they are attached.

Adverbials formed from phrases and clauses are preceded by a preposition or subordinate conjunction. These prepositions and conjunctions are used as literal role markers to determine the correct relationship by which the adverbial graphs should be attached. The structure modified by the adverbial calls the adverbial rules which process the phrase or clause. The graph of the adverbial phrase or clause is then passed back with the literal role marker that precedes the phrase or clause. The structure that is modified then uses the selection process of the attach operation to determines the conceptual relation by which the adverbial should be attached. Examples of adverbial processing can be found in the detailed example given in Section 2 of Chapter 6 which contains two adverbial structures.

5.6.3. Head Rules

Head rules in the semantic analyzer are defined for the processing of nouns and compound nouns which are often called nominal compounds. The simplest of head structures are made up of individual nouns and identifiers. When a head structure is made up of a single noun, the root word for the noun is retrieved from the database. The canonical conceptual graph is then retrieved. Identifiers are made up of names, such as acronyms, that are not in the ASPI system dictionary. Therefore the Semantic Analyzer creates a graph for the identifier designating ID as its type. In order to handle
identifier types during the attachment of graphs, a special rule exists. When an identifier type graph is to be attached to another conceptual graph, the special rule checks the type of graph that is required to make the attachment. If the type required for attachment is a subtype of OBJECT, then attachment is made, since identifiers represent objects such as values and devices. For example, the sentence

_The data is loaded into the register when STRB rises_

contains the word STRB, which is not in the ASPIN dictionary. During analysis, a conceptual graph is created for STRB with the type ID. The verb *rise* has a canonical graph, as shown in Figure 23 (Chapter 6), with the role *object* that can be filled by a subtype of VALUE. In order for the attachment to be made, VALUE must be a subtype of OBJECT. Since VALUE is a valid subtype, the attachment is made and STRB is attached to rises using the conceptual relation *object*.

Nominal compounds used in English are difficult to analyze. Therefore, the Semantic Analyzer has several rules to handle such structures. An example of a nominal compound can be seen in the sentence,

_The CPU register is cleared._

*CPU register* is a nominal compound representing a single entity, but is made up of two nouns; *CPU* and *register*.
There are seven rules for these type of compounds. Analysis will test the given nominal compound against the set of rules until a valid rule is found. For example, the analysis of the nominal compound given in the sentence above would complete when the rule head3a is attempted. Rule head3a tests the two parts of the compound to verify that they are both subtypes of objects. If so, the Semantic Analyzer tries to attach the first noun in the compound to the second using the relation contain. Since CPU and register are both subtypes of OBJECT, an attempt is made to attach the graph of CPU to the graph of register using the conceptual relation in. The graph for the above example is given below.
Nominal compounds are often made up of two words in which one is a more general term for the other. Examples of this can be seen in *microcontroller unit* and *system memory*. The analyzer contains two specific rules for such cases. The rules find the more specific type of the two words and concatenates the names to form a compound word. For example, in *microcontroller unit*, *microcontroller* is a subtype of *PROCESSOR*, which is more specialized than *unit*, which is a subtype of *DEVICE*. The resulting concept is formed as follows:

[ 1 : MICROCONTROLLER : microcontroller_unit ]

A *deverbal* noun is a verb in noun form. When paired with another noun, a degenerate clause is formed, as in *a memory access*. Degenerate clauses are common in English and require special handling from the semantic analyzer. In the noun phrase *a memory access*, which signifies an *access* of *memory*, *memory* modifies *access*. The semantic analyzer handles cases like this by examining the graph of the second noun. If the concept is a subtype of *ACTION*, then the analyzer tries to attach the canonical graph of the first noun to the canonical graph of the second noun using the conceptual relation *object*. In this example, *memory* would be attached as the object of *access* as shown below.
Another example of a degenerate clause can be seen in the noun phrase *the reset line*. In this case the first noun is a subtype of ACTION and the second noun is of type CARRIER. The analyzer handles such cases by attaching the graph of the ACTION type noun to the graph of the CARRIER type noun using the relation *purpose*. Thus, in this example, the canonical graph of *reset* is attached to *line* using the relation *purpose* as shown below.

```
[ 1 : LINE : line ]
- > ( purp ) - > [2]
[ 2 : RESET : reset ]
```

5.6.4. Nominal Rules

The nominal rules in the semantic analyzer define the methods of attachment for the conceptual graphs of noun phrases and prepositional phrases which make up the nominals in a sentence. Nominals can be formed from several different combinations of grammatical structures. Therefore, the semantic rules define eleven unique nominal rules.

Nominals are often formed by a noun phrase followed by a prepositional phrase. In such cases, the prepositional phrase modifies the noun phrase. Therefore, the conceptual graph of the prepositional phrase it attached to the graph of the noun phrase using the preposition as a literal role marker for the attachment. For example, the nominal *the*
*clock on the motherboard* consists of a noun phrase (*the clock*) followed by a prepositional phrase (*on the motherboard*). The graphs are attached using the literal role marker *on*. *On* matches the role marker required to attach to the relation *location* as defined in the canonical graph of *clock*. Thus the two graphs are attached as shown below.

\[
[1 : \text{CLOCK : clock} ] \\
\quad \rightarrow (\text{loc : on}) \quad \rightarrow [2] \\
\quad \rightarrow (\text{det : the}) \\
[2 : \text{MOTHERBOARD : motherboard} ] \\
\quad \rightarrow (\text{det : the})
\]

Nominals are often formed by noun phrases linked with the preposition *of*. There are typically two types of nominals that fit this category. The first, has an action as the first noun phrase, as in *execution of the instruction*. The second, has a description of measure in the first noun phrase, as in *the byte of memory*. For the first type of nominal, the second noun phrase is attached to the action noun phrase using the conceptual relation *agent* or *object* as shown in the example below.

\[
[1 : \text{EXECUTE : execute} ] \\
\quad \rightarrow (\text{obj}) \quad \rightarrow [2] \\
[2 : \text{INSTRUCTION : instruction} ] \\
\quad \rightarrow (\text{det : the})
\]

For the second type of nominal, the graph of the first noun phrase is checked to verify it is a subtype of *MEASURE*. If so, the graph of the first noun phrase is attached to the graph of the second noun phrase using the conceptual relation *size*. Below is the graph for *byte of memory*. 
[ 1 : MEMORY : memory ]
    - > ( size )    - > [ 2 ]
[ 2 : MEM_MEASURE : byte ]

5.6.5. Predicate Rules

The predicate rules of the semantic analyzer attach the graphs of the subject and objects of the sentence to the graphs of the verbs. There are three types of verb sequences used in the ASPIN English sublanguage: active verb sequences, passive verb sequences, and attributive verb sequences. Active verb sequences represent active voice sentences. In active voice, the subject of the sentence does the action described by the verb. In passive voice, the subject is the recipient of the action. Therefore, a structural role marker is used to attach the subject to the verb based on the type of voice used in the sentence. The sentence the register is reset is in passive voice. Therefore the predicate rule attaches the graph of the subject, register, to the graph of the verb, reset, using the structural role marker object. The Semantic Analyzer examines the graph of reset and searches for a slot that contains the structural role marker object. When the slot is found, the analyzer attempts to attach the graph of the subject using the relation designated by the slot. This can be illustrated using the example above. A canonical conceptual graph for reset is given below.

[ RESET : reset ]
    - > (obj : obj ) - > [ MEMORY ]
    - > (agnt : {by | agnt]) - > [ DEVICE ]
Since the register is reset is in passive voice, the analyzer tries to attach the graph of register to the graph of reset using the structural role marker object. According to the canonical conceptual graph for reset, a subtype of MEMORY is required to be attached using the relation obj. Since register is a subtype of MEMORY, the attachment is successful.

The predicate rules defined for the attachment of active voice verb sequences is similar. The attachment is also based on the a structural role marker, but the since the subject of active voice sentences is the actor of the verb, the structural role marker agent is used.

Attributive verb sequences represent states of being, as in the sentence the signal is high. These types of verb sequences can be followed by adverbials, nominals, or adjectives. In the counter is in up/down mode, in up/down mode is an adverbial phrase following the stative verb is. In the counter is a ripple-carry device, the attributive verb sequence is followed by the nominal a ripple-carry device. An example of an adjective following an attributive verb sequence can be seen in the signal is high, given above. The predicate rules defined for attributive verb sequences attach the subject of the sentence using the relation agent. Nominals are attached using the relation object. Other structures such as adjectives and adverbials are attached according to the rules defined for those structures.

5.6.6. Conjunction Processing

Conjunctions can be used to join multiple predicates, nominals, and simple sentences. The use of conjunctions presents some difficulty since the processing of conjoined structures requires a resultant conceptual type that can be used to for later attachments.
The semantic analyzer handles these situations by determining the minimal common supertype of the graphs of the conjoined structures. The minimal common supertype then becomes the type of concept representing the conjoined structures. An example of this process can be seen in the sentence *the machine can perform a read or write asynchronously*. The minimal common supertype of READ and WRITE is MOVE. Therefore the graphs for *read* and *write* are attached to a conjunction concept which is given the type MOVE, as shown below.

\[
[ 3 : MOVE : and ]
\quad - > ( and : or ) \quad - > [ 2 ]
\quad - > ( and : or ) \quad - > [ 3 ]
\quad [ 4 : READ : read ]
\quad [ 5 : WRITE : write ]
\]

The resulting conceptual graph for the sentence is shown below.

\[
[ 1 : PERFORM : perform ]
\quad - > ( agnt ) \quad - > [ 2 ]
\quad - > ( obj ) \quad - > [ 3 ]
\quad - > ( man : adv ) \quad - > [ 4 ]
\quad - > ( mod : can )
\quad [ 2 : MACHINE : machine ]
\quad - > ( det : the )
\quad [ 3 : MOVE : and ]
\quad - > ( and : or ) \quad - > [ 5 ]
\quad - > ( and : or ) \quad - > [ 6 ]
\quad [ 4 : ACT_STRUCT : asynchronously ]
\quad [ 5 : READ : read ]
\quad [ 6 : WRITE : write ]
\]
5.7. Inferencing

Metonymy occurs when something is referred to using its characteristics. For example, in the sentence the byte is incremented by the counter, the word byte is a subtype of MEMORY_MEASURE. Counters increment values rather than MEMORY_MEASURE types. Thus, the value incremented by the counter is referred to using its size. Analysis of sentences is often made difficult by the metonymy that occurs in natural language.

To handle cases of metonymy, the semantic analyzer uses a set of inference facts that can be referred to during the attach operation. The facts are based on inference made between the type of the concept being attached and the type required by the empty slot of the head graph. In the byte is incremented by the counter, given above, the inference facts would indicate that a value can be contained in a subtype of MEMORY_MEASURE. Therefore, when the type constraints of the empty slot are checked against the conceptual type of the attaching graph, the analyzer knows that byte can represent a subtype of VALUE.

The semantic analyzer makes use of the rules when the concept type of the attaching graph does not match the type constraints set by the empty slot of head graph. The analyzer then tries inference based on the conceptual type of the attaching graph. This is done by searching for an inference fact that describes a relationship between the type of the attaching graph and the required type of the empty slot. If such an inference fact exists, then a new conceptual graph is created. The conceptual type of this new graph is the type required by the empty slot. The attaching graph is joined to this new conceptual graph using the relation specified in the inference fact. This joined graph is then attached.
to the empty slot of the head graph. The conceptual graph resulting from the inference in
the byte is incremented by the counter is shown below.

\[
[1: \text{INCREMENT} : \text{increment} ]
- > ( \text{agt} : \text{by} )
- > [2]
- > ( \text{obj} )
- > [3]
[2: \text{COUNTER} : \text{counter} ]
[3: \text{VALUE} : \ast ]
- > ( \text{size} )
- > [4]
[4: \text{BYTE} : \text{byte} ]
\]

The canonical graph of increment requires a VALUE subtype to fill the role of object. Since byte is a subtype of MEMORY_MEASURE and not a VALUE type, a search of the inference rules is made in order to find a fact that relates a MEMORY_MEASURE subtype to a VALUE subtype. An appropriate inference fact is found and a new conceptual graph is created with the type VALUE. The referent of the VALUE type concept is the generic referent \('*', and the concept byte is attached to VALUE using the conceptual relation size as specified by the inference fact that was used. This graph is then attached to increment using the conceptual relation object.
Chapter 6. Analysis Examples

6.1. Introduction

Included in this chapter are several examples taken from the suite of sentences used to test all of the semantic rules. The first example contains a detailed discussion of its analysis. The rest of the examples show some of the capabilities of the semantic analyzer.

6.2. A Detailed Example

The first example given is a detailed analysis for the sentence the 8-bit data is loaded into the ACC register when STRB rises, which will be referred to as sentence #1.

The sentence must first be parsed the according to the English grammar rules defined in Appendix B. The dictionary is searched for the lexical categories of each word in the sentence. Table 5 shows the grammatical categories for each of the words in sentence #1. All the words in sentence #1 are contained in the dictionary except ACC and STRB.
Since these two words are not contained in the dictionary, they are assumed to be identifiers and are therefore given the category id.

**Table 5: Grammatical Categories for the Words in Sentence #1**

<table>
<thead>
<tr>
<th>Word</th>
<th>Grammatical Category</th>
</tr>
</thead>
<tbody>
<tr>
<td>the</td>
<td>det</td>
</tr>
<tr>
<td>8-bit</td>
<td>adj</td>
</tr>
<tr>
<td>data</td>
<td>noun</td>
</tr>
<tr>
<td>is</td>
<td>be</td>
</tr>
<tr>
<td>loaded</td>
<td>ven</td>
</tr>
<tr>
<td>into</td>
<td>scon</td>
</tr>
<tr>
<td>the</td>
<td>det</td>
</tr>
<tr>
<td>ACC</td>
<td>id</td>
</tr>
<tr>
<td>register</td>
<td>noun</td>
</tr>
<tr>
<td>when</td>
<td>scon</td>
</tr>
<tr>
<td>STRB</td>
<td>id</td>
</tr>
<tr>
<td>rises</td>
<td>verb</td>
</tr>
</tbody>
</table>

These lexical categories shown in Table 5 are used to build up the syntactic parse tree for sentence #1, which is shown in Figure 22. At the root of the parse tree is the rule s1 which states that a sentence can be formed by a simple sentence (ss) followed by a period. The leaves of the tree correspond to the words used in the sentence. This parse tree is used as the input to the semantic analyzer.

The semantic analyzer traverses the input parse tree from top to bottom. Each node of the parse tree corresponds to a Prolog rule in the semantic analyzer. Processing begins at the root of the parse tree and continues along each subtree of each node. The tree is traversed from the top branches to the bottom branches. Processing continues until a leaf node is reached. At this point, a graph is constructed. If the leaf node in question is a
type of noun, verb, or adjective, a canonical graph is retrieved from the canon. Otherwise, an appropriate graph is created, as is the case of identifiers, such as STRB and ACC in sentence #1.

Figure 22: Syntactic Parse Tree for Sentence #1

Figure 23 shows some of the canonical conceptual graphs for the words in sentence #1. According to semantic rule np11, the graph of the adjective 8-bit should be attached to the graph of data. As shown in Figure 23, the adjective 8-bit can be attached to a type of PROCESSOR or VALUE using the relation size. Therefore, since DATA is a subtype of VALUE, the attachment is made. The resulting graph is shown in Figure 24.
Figure 23: Canonical Graphs for "load," "data," "rise," and "8-bit"

Figure 24: Conceptual Graph of Sentence #1 After np11 is Processed

The semantic analyzer always passes the graph of the subject of the sentence forward to the predicate when it is processed. Therefore, the subject graph is attached to the verb graph during the processing of the predicate.
The predicate of sentence #1 consists of a passive verb sequence followed by two adverbials. Semantic rule pred9 processes the passive verb sequence first, followed by the two adverbial phrases.

Processing of the passive verb sequence is accomplished by finding the root word of loaded and retrieving the canonical graph for that root word. The database of root words is consulted and load is returned as the root of loaded. The semantic analyzer then retrieves the canonical graph for load and passes it back to the predicate procedure for later attachments. Figure 25 illustrates this stage of the analysis.

Figure 25: Conceptual Graph of Sentence #1 After pvs1 is Processed
After the verb sequence has been processed, the adverbial phrases are processed. When this is finished, the semantic analyzer attempts to attach the graph of the subject of the sentence to the graph resulting from the passive verb sequence. Since the predicate is in passive voice, the graph of the subject of the sentence is attached using the structural role marker object. The required conceptual type to fill the role of object of the verb load is VALUE. Since DATA is a subtype of VALUE, the attachment succeeds. Figure 26 shows how the conceptual graph looks at this stage in the processing.

![Conceptual Graph of Sentence #1 After Subject is Attached to Predicate](image)

**Figure 26: Conceptual Graph of Sentence #1 After Subject is Attached to Predicate**

After the subject and verb sequence graphs have been attached, the semantic analyzer attempts to attach the graphs of the adverbial phrases to verb graph. The first adverbial graph contains the canonical graph of register. The identifier ACC has also been
attached to it using the conceptual relation *name*. In order to attach this graph to the verb
graph, the literal role marker *into* is used. According to the canonical graph of *load*, the
literal role marker *into* indicates that the graph of the adverbial phrase can be joined to
the graph of *load* using the conceptual relation *destination*. In order to fill this relation
the attaching concept must be a DEVICE or subtype of DEVICE. Since REGISTER is a
subtype of DEVICE, the attachment succeeds.

Attachment of the second adverbial phrase to the graph of *load* is based on the literal role
marker *when*. The role marker *when* indicates during the attach operation that the graph
of the second adverbial phrase should be attached using the conceptual relation *condition*.
The completed conceptual graph for sentence #1 is shown in Figure 27.

![Complete Conceptual Graph of Sentence #1](image)

**Figure 27: Complete Conceptual Graph of Sentence #1**

### 6.3. Other Analysis Examples

This section contains five more examples of conceptual graphs created for sentences
taken from the suite used to test the semantic rules of the analyzer. The first sentence
analyzed in this section is *the 8048 has 27 lines which can be used for input functions or output functions*. The main part of the sentence is *the 8048 has 27 lines*. The restrictive
relative clause which can be used for input functions or output functions modifies lines.
The semantic analyzer handles restrictive relative clauses by creating a separate conceptual graph that includes the word modified by the clause. Thus, the modified word appears in both of the graphs and is a concept by which the two conceptual graphs for this sentence would be joined in post-processing. A post-processor, which is discussed in Chapter 7, has not yet been developed.

The main verb of the sentence is have. 8048 is not in the ASPIN dictionary, so it is classified as an identifier. The sentence is in active voice, so the analyzer attaches the graph of 8048 to the graph of have using the structural role marker agent. The relative clause contains the conjunction or, which causes the semantic analyzer to create a concept representing the conjunction of input functions and output functions. The type for the conjunction concept is the minimal common supertype of the conjoined structures, which is FUNCTION. The nominal compounds of the relative clause, namely input functions and output functions, use one of the head rules in the semantic analyzer. Since the first part of the nominal compound is a subtype of ACTION, its graph is attached to the rest of the nominal compound using the conceptual relation purpose.

the 8048 has 27 lines which can be used for input functions or output functions.

Main Conceptual Graph
[
  [ 1 : have : have ]
  ->( agnt : agnt ) -> [ 2 ]
  ->( obj : obj ) -> [ 3 ]
  [ 2 : id : 8048 ]
  ->( det : the )
  [ 3 : line : line ]
Relative Conceptual Graph

[ 1 : use : use ]
  ->( mod : can )
  ->( obj : obj ) -> [ 2 ]
  ->( purp : for ) -> [ 3 ]
[ 2 : line : line ]
  ->( quant : null ) -> [ 4 ]
[ 4 : # : 27 ]
[ 3 : function : or ]
  ->( or : null ) -> [ 5 ]
  ->( or : null! ) -> [ 6 ]
[ 5 : function : function ]
  ->( purp : null ) -> [ 7 ]
[ 7 : input_1 : input ]
[ 6 : function : function ]
  ->( purp : null ) -> [ 8 ]
[ 8 : output_1 : output ]

This graph also shows a limitation of the semantic analyzer. The words input and output can have several different uses. This results in the classification of the same word in different parts of the hierarchy. In order to be able to distinguish which sense of the word is used in the sentence, such words are given a number that serves an identifier. The number is concatenated to the word using an underscore, such as input_1, or output_2. This allows for proper interpretation for the other parts of the ASPIN system that use the conceptual graphs as input.
The second conceptual graph example is for the sentence *a memory write stores the data.* This sentence contains a nominal compound, *memory write*, that acts as the subject of the sentence. Since *memory* is a subtype of OBJECT and *write* a subtype of ACTION, *memory* is attached to *write* using the conceptual relation *destination*, which is designated in the canonical graph of *write*. The graph for this sentence is shown below.

*a memory write stores the data.*

```
[ 1 : store : store ]
 ->( agnt : agnt )  -> [ 2 ]
 ->( obj : obj )     -> [ 3 ]
[ 2 : write : write ]
 ->( dest : to )     -> [ 4 ]
 ->( det : a )
[ 4 : memory : memory ]
[ 3 : data : data ]
 ->( det : the )
```

The third example sentence in this section is *these bits can be tested by a program and specific action can be taken as a result of their state.* This is a compound sentence that requires inference by the semantic analyzer. The verb *test* requires a DATA type or CARRIER type to fill the role of object. *Bits*, which is in the structural position to fill that role, is a subtype of MEMORY_MEASURE. Since MEMORY_MEASURE is the incorrect type to fill the desired role, the semantic analyzer searches the set of inference rules for a fact relating MEMORY_MEASURE types to DATA types. One exists that states that DATA types are located in MEMORY_MEASURE types. The resulting graph is shown below.
These bits can be tested by a program and specific action can be taken as a result of their state.

[ 1 : action : and ]
->( and : null )   -> [ 2 ]
->( and : null )   -> [ 3 ]
[ 2 : test : test ]
->( mod : can )
->( obj : obj )    -> [ 4 ]
->( agnt : by )    -> [ 5 ]
[ 4 : data : * ]
->( size : null )  -> [ 6 ]
[ 6 : bit : bit ]
->( det : these )
[ 5 : program : program ]
->( det : a )
[ 3 : take : take ]
->( mod : can )
->( obj : obj )    -> [ 7 ]
->( cond : as a result of) -> [ 8 ]
[ 7 : action : action ]
->( attr : adj )    -> [ 9 ]
[ 9 : characteristic : specific ]
[ 8 : state : state ]
->( det : their )

The two simple sentences are joined to the conjunction concept and, which takes the minimal common supertype of the verbs of the simple sentences. The phrase as a result also occurs in this sentence. The semantic analyzer recognizes the subtree of as a result to be a literal role marker, which results in the graphs of a specific action can be taken and their state being linked together using the condition relation.
The fourth example is the graph for *the program counter is a 16-bit register that contains the address of the next instruction to be executed*. The main clause of the sentence is *the program counter is a 16-bit register*. The rest of the sentence is made up of two restrictive relative clauses. Therefore, the analysis of this sentence produces three conceptual graphs, which are shown below.

*The program counter is a 16-bit register that contains the address of the next instruction to be executed.*

**Main Conceptual Graph**

[ 1 : is : is ]
  ->( agnt : agnt )  -> [ 2 ]
  ->( obj : obj )    -> [ 3 ]

[ 2 : counter : counter ]
  ->( type : null ) -> [ 4 ]
  ->( det : the )

[ 4 : program : program ]

[ 3 : register : register ]
  ->( type : adj ) -> [ 5 ]
  ->( det : a )

[ 5 : mem_measure : 16-bit ]

**Relative Conceptual Graphs**

[ 1 : execute : execute ]
  ->( obj : obj ) -> [ 2 ]

[ 2 : instruction : instruction ]
  ->( attr : null ) -> [ 3 ]
  ->( det : the )

[ 3 : next : next ]

[ 1 : contain : contain ]
The fifth example sentence is *execution of a startcnt instruction connects the t1 input pin to the counter input and enables the counter*. This example uses inference to attach the graph of *counter* to the graph of *enable*. The canonical graph for *enable* requires a type of *ACTION* to fill the role of *object*. *Counter* is a subtype of *DEVICE* and cannot fill the *object* role of *enable*. The semantic analyzer therefore searches and finds an inference fact that states that counters can be the agent of an *ACTION*. This sentence also contains two different uses of the word *input*. In *input pin*, *input* represents the action input. In *counter input*, *input* represents the carrier sense of input. These different senses are labeled with numbers, as discussed earlier in this section. The graph for the sentence is shown below.

*Execution of a startcnt instruction connects the t1 input pin to the counter input and enables the counter.*

```
[ 1 : universal : and ]
->( and : null ) -> [ 2 ]
```
->( and : null ) -> [ 3 ]
[ 2 : connect : connect ]
->( agnt : agnt ) -> [ 4 ]
->( obj : obj ) -> [ 5 ]
->( loc : to ) -> [ 6 ]
[ 4 : execute : execute ]
->( obj : obj ) -> [ 7 ]
[ 7 : instruction : instruction ]
->( name : null ) -> [ 8 ]
->( det : a )
[ 8 : id : startcnt ]
[ 5 : pin : pin ]
->( purp : null ) -> [ 9 ]
->( name : null ) -> [ 10 ]
->( det : the )
[ 9 : input_1 : input ]
[ 10 : id : t1 ]
[ 6 : input_2 : input ]
->( type : null ) -> [ 11 ]
->( det : the )
[ 11 : counter : counter ]
[ 3 : enable : enable ]
->( agnt : agnt ) -> [ 12 ]
->( obj : obj ) -> [ 13 ]
[ 12 : execute : execute ]
->( obj : agnt ) -> [ 14 ]
[ 14 : instruction : instruction ]
->( name : null ) -> [ 15 ]
->( det : a )
[ 15 : id : startcnt ]
[ 13 : action : * ]
->( agnt : null ) -> [ 16 ]
[ 16 : counter : counter ]
->( det : the )
The final example is the sentence the \textit{m6811 is an 8-bit, hcmos, microcontroller unit (mcu)}. This sentence contains the nominal compound \textit{microcontroller unit}, in which \texttt{MICROCONTROLLER} is a subtype of \texttt{DEVICE}. During processing, the two words are concatenated together and given the more specific type. This sentence also contains a parenthetical name \textit{mcu}. The semantic analyzer handles such structures by attaching the graph of the concept inside the parentheses to the graph of the noun phrase immediately preceding using the conceptual relation \textit{name}. The graph for the sentence is given below.

\begin{quote}
\textit{The m6811 is an 8-bit, hcmos microcontroller unit (mcu).}
\end{quote}

\begin{verbatim}
[ 1 : is : is ]
  ->( agnt : agnt )  -> [ 2 ]
  ->( obj : obj )     -> [ 3 ]
[ 2 : m6811 : m6811 ]
  ->( det : the )
[ 3 : microcontroller : microcontroller_unit ]
  ->( type : adj )   -> [ 4 ]
  ->( attr : adj )   -> [ 5 ]
  ->( det : an )
  ->( name : null )  -> [ 6 ]
[ 4 : mem_measure : 8-bit ]
[ 5 : characteristic : hcmos ]
[ 6 : id : mcu ]
\end{verbatim}
Chapter 7. Future Work

7.1. System Capabilities

The semantic analyzer is able to handle a large number of system description sentences. This system can handle simple sentences, compound sentences, and sentences with compound predicates. In all, the grammar contains over 115 productions using 48 non-terminal structures. The canon contains approximately 380 canonical graphs that cover usage of over 680 words. Not included in the count are words such as determiners and prepositions, which do not have canonical graphs.

7.2. System Limitations

Words in the vocabulary sometimes have multiple meanings that require different canonical graphs. This is called lexical ambiguity [8]. "Input", for example can be used as a verb in "The data is input to the processor", a noun in "The input is sent to the processor", or an adjective in "The input data is sent to the processor." Sentences containing words with multiple meanings will often have several syntactically valid parse trees. Although the trees are structurally correct, most have no valid meaning. The
semantic analyzer will reject nonsensical trees in most cases due to the type constraints built into the slots of the canonical graphs. Work needs to be done to resolve cases where multiple valid graphs are generated. Elements have already been added to the data structure of the analyzer to included a number which can be used as a confidence level for a given conceptual graph. Routines have also been added that analyze all the parse trees in an input file and compares the valid graphs generated. The analyzer will report the one most likely to be correct based on the level of confidence assigned to each graph. Although the data structure exists as well as the routine to do the analysis, a scheme calculating confidence levels has not yet been implemented.

Although the grammar can handle most types of system description sentences, there are a few structures that cannot be interpreted. Semicolons and colons are currently not recognized by the parser and semantic analyzer.

Recently, an expanded dictionary of over 1200 words was introduced for use by the parser. Currently, the semantic analyzer does not contain canonical graphs for all the words in this new dictionary.

7.3 Directions for Future Work

The semantic analyzer handles one sentence at a time. A set of graphs for a given description must be joined and simplified manually in order to come up with a conceptual graph representing the entire system being described. The development or modification of an existing conceptual graph processor would greatly expand the usefulness of the semantic analyzer.
As discussed in the previous section, a scheme for computing confidence levels associated with conceptual graphs needs to be investigated. One possible method might be based on the relative frequency of grammatical structures used in descriptions of digital systems. Two conceptual graphs for the same sentence might be able to be evaluated based on which input parse tree contains the more common grammatical structures. A conceptual graph produced from a parse tree containing uncommon structures could be disregarded when compared with a graph created from a tree containing common structures. Another possible approach is to use a conceptual graph processor in post-processing to join the combinations of conceptual graphs generated for several sentences. The best graph could be determined to be the graph containing the fewest concepts [8]. Since the lower the number of concepts indicates the greater number of joins, it seems likely that most compact conceptual graph will contain the most accurate interpretation of the system description.

Currently, the search for the root of a word is done by the semantic analyzer. Thus, when a new word is introduced into the vocabulary of the system, both the database of canonical graphs and the database of root words must be modified. It would be more efficient to include the root form of a word in its dictionary entry that is read by the parser. The parser could then substitute the root form of the word in the parse tree for use by the semantic analyzer. This method would be more efficient both computationally and in terms of database management. Including the root word in the parse tree allow the semantic analyzer to skip the search for root word forms currently required, thus cutting the number of word searches done by the parser and semantic analyzer in half. Also,
introduction of new words into the vocabulary would require just one semantic analyzer
database change.

Work needs to be done to expand the number of canonical graphs of the system in order
to cover all of the words in the recently introduced expanded dictionary. Completion of
this task would allow seamless integration of the parser and semantic analyzer.
Chapter 8. Instructions for the Semantic Analyzer

As was mentioned in Chapter 1, the Semantic Analyzer software is written in Quintus Prolog on a Sun Sparcstation platform. The software consists of five Prolog files: main.pl, semrules.pl, canon.pl, roots.pl, and suite.pl. The first four are the core files, and the last file contains the correct parse trees for the suite of test sentences. The Quintus Prolog for the Sun Platform can be run in a standard UNIX shell or using an X-Windows interface. To enter the Prolog from a UNIX shell, at the prompt type:

```
prolog
```

To enter Prolog using an X-Windows interface, use the command:

```
qui
```

Once in Prolog, you should receive the standard Prolog prompt `?-`. To run the semantic analyzer, enter the following command:

```
?- consult(main.pl).
```
This command will compile all the semantic analyzer files. The suite of test sentences is numbered 1-70. To process a particular sentence, sentence number 6 for example, type the following command:

?- proc(6).

The semantic analyzer will then traverse the parse tree and create the conceptual graph for the sentence.

A multiple sentence processing routine also exists. This routine will process every sentence in the input file, displaying each graph as it is created. When all the sentences have been processed, the system will display the graph with the highest confidence level. Since the methodology for determining conceptual graph confidences has not been implemented, the first conceptual graph successfully created will be repeated at the end of processing. To invoke the multiple sentence processing routine, enter the following command:

?- go.

To quit the semantic analysis session, type the following at the Quintus Prolog prompt:

?- quit.
Bibliography


Appendix A. Vocabulary

This appendix contains the list of words for which canonical conceptual graphs are defined.

<table>
<thead>
<tr>
<th>Word</th>
<th>Accumulators</th>
<th>Acknowledge</th>
<th>Affect</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>accept</td>
<td>affect</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>accepted</td>
<td>affects</td>
<td></td>
</tr>
<tr>
<td>16-bit</td>
<td>accepting</td>
<td>allow</td>
<td></td>
</tr>
<tr>
<td>3-bit</td>
<td>accepts</td>
<td>allowed</td>
<td></td>
</tr>
<tr>
<td>3-input</td>
<td>access</td>
<td>allows</td>
<td></td>
</tr>
<tr>
<td>8-bit</td>
<td>accessed</td>
<td>alter</td>
<td></td>
</tr>
<tr>
<td>8k</td>
<td>accessing</td>
<td>altered</td>
<td></td>
</tr>
<tr>
<td>accept</td>
<td>accomplish</td>
<td>alters</td>
<td></td>
</tr>
<tr>
<td>accepted</td>
<td>accomplished</td>
<td>alu</td>
<td></td>
</tr>
<tr>
<td>accepting</td>
<td>accomplishes</td>
<td>alus</td>
<td></td>
</tr>
<tr>
<td>accepts</td>
<td>accomplishing</td>
<td>and</td>
<td></td>
</tr>
<tr>
<td>access</td>
<td>action</td>
<td>anded</td>
<td></td>
</tr>
<tr>
<td>accessed</td>
<td>activate</td>
<td>ands</td>
<td></td>
</tr>
<tr>
<td>accesses</td>
<td>activated</td>
<td>apply</td>
<td></td>
</tr>
<tr>
<td>accessing</td>
<td>activates</td>
<td>application</td>
<td></td>
</tr>
<tr>
<td>accomplish</td>
<td>act</td>
<td>applied</td>
<td></td>
</tr>
<tr>
<td>accomplished</td>
<td>added</td>
<td>applies</td>
<td></td>
</tr>
<tr>
<td>accomplishes</td>
<td>add</td>
<td>are</td>
<td></td>
</tr>
<tr>
<td>accomplishing</td>
<td>addend</td>
<td>argument</td>
<td></td>
</tr>
<tr>
<td>action</td>
<td>addends</td>
<td>arguments</td>
<td></td>
</tr>
<tr>
<td>actions</td>
<td>adder</td>
<td>array</td>
<td></td>
</tr>
<tr>
<td>accumulate</td>
<td>adders</td>
<td>arrays</td>
<td></td>
</tr>
<tr>
<td>accumulated</td>
<td>addition</td>
<td>arrival</td>
<td></td>
</tr>
<tr>
<td>accumulates</td>
<td>address</td>
<td>arrive</td>
<td></td>
</tr>
<tr>
<td>accumulating</td>
<td>addressed</td>
<td>arrived</td>
<td></td>
</tr>
<tr>
<td>accumulator</td>
<td>addresses</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>adds</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
arrives  caching  component
assembler  call  components
assemblers  called  compute
assert  calling  computed
asserted  calls  computer
asserting  cause  computers
assertion  caused  computes
asserts  causes  computing
asynchronous  causing  conditional
asynchronously  change  connect
attempt  changed  connected
attempted  changing  connecting
attempting  chip  connects
attempts  chips  contain
average  circuit  contained
averaged  circuitry  containing
averages  circuits  contains
averaging  clear  content
augend  cleared  contents
augends  clearing  continue
be.is  clears  continued
began  clock  continues
begin  clocked  continuing
beginning  clocking  control
begins  clocks  controlled
begun  code  controllers
bit  codes  controlling
bits  coincide  controls
branch  coincided  conversion
branched  coincides  convert
branches  coinciding  converted
branching  command  converting
buffer  commands  converts
collectors  compare  co-processor
collectors  compared  co-processors
count  compares  count
counts  comparing  counters
couter:  complete  cpu
cubes  completes  data
cache  completing  decode
cached  completion
<table>
<thead>
<tr>
<th>parameters</th>
<th>pushing</th>
<th>reset</th>
</tr>
</thead>
<tbody>
<tr>
<td>pass</td>
<td>quotient</td>
<td>resets</td>
</tr>
<tr>
<td>passed</td>
<td>quotients</td>
<td>resetting</td>
</tr>
<tr>
<td>passes</td>
<td>ram</td>
<td>result</td>
</tr>
<tr>
<td>passing</td>
<td>read</td>
<td>results</td>
</tr>
<tr>
<td>perform</td>
<td>reading</td>
<td>resume</td>
</tr>
<tr>
<td>performance</td>
<td>reads</td>
<td>resumed</td>
</tr>
<tr>
<td>performed</td>
<td>receive</td>
<td>resumes</td>
</tr>
<tr>
<td>performing</td>
<td>received</td>
<td>resuming</td>
</tr>
<tr>
<td>performs</td>
<td>receives</td>
<td>return</td>
</tr>
<tr>
<td>peripherals</td>
<td>receiving</td>
<td>returned</td>
</tr>
<tr>
<td>period</td>
<td>record</td>
<td>returning</td>
</tr>
<tr>
<td>periods</td>
<td>recorded</td>
<td>returns</td>
</tr>
<tr>
<td>pin</td>
<td>rise</td>
<td>rise</td>
</tr>
<tr>
<td>pins</td>
<td>rises</td>
<td>rises</td>
</tr>
<tr>
<td>point</td>
<td>rom</td>
<td>routine</td>
</tr>
<tr>
<td>pointed</td>
<td>routine</td>
<td>run</td>
</tr>
<tr>
<td>pointing</td>
<td>routines</td>
<td>running</td>
</tr>
<tr>
<td>points</td>
<td>run</td>
<td>runs</td>
</tr>
<tr>
<td>poll</td>
<td>save</td>
<td>save</td>
</tr>
<tr>
<td>polled</td>
<td>saved</td>
<td>saves</td>
</tr>
<tr>
<td>polling</td>
<td>saves</td>
<td>saving</td>
</tr>
<tr>
<td>polls</td>
<td>saving</td>
<td>selector</td>
</tr>
<tr>
<td>port</td>
<td>selector</td>
<td>selectors</td>
</tr>
<tr>
<td>ports</td>
<td>send</td>
<td>send</td>
</tr>
<tr>
<td>priority</td>
<td>sending</td>
<td>sends</td>
</tr>
<tr>
<td>procedure</td>
<td>sends</td>
<td>sensor</td>
</tr>
<tr>
<td>procedures</td>
<td>sensors</td>
<td>services</td>
</tr>
<tr>
<td>processor</td>
<td>sent</td>
<td>service</td>
</tr>
<tr>
<td>processors</td>
<td>service</td>
<td>serviced</td>
</tr>
<tr>
<td>product</td>
<td>services</td>
<td>servicing</td>
</tr>
<tr>
<td>products</td>
<td>set</td>
<td>set</td>
</tr>
<tr>
<td>program</td>
<td>setting</td>
<td>sets</td>
</tr>
<tr>
<td>programs</td>
<td>sets</td>
<td>shift</td>
</tr>
<tr>
<td>prom</td>
<td>shift</td>
<td>shifted</td>
</tr>
<tr>
<td>proms</td>
<td>shift</td>
<td>shifting</td>
</tr>
<tr>
<td>pulse</td>
<td>shift</td>
<td>shifting</td>
</tr>
<tr>
<td>pulsed</td>
<td>shift</td>
<td>shifting</td>
</tr>
<tr>
<td>pulses</td>
<td>shift</td>
<td>shifting</td>
</tr>
<tr>
<td>push</td>
<td>shift</td>
<td>shifting</td>
</tr>
<tr>
<td>pushed</td>
<td>shift</td>
<td>shifting</td>
</tr>
<tr>
<td>pushes</td>
<td>shift</td>
<td>shifting</td>
</tr>
</tbody>
</table>
Appendix B. Grammar Rules and Abbreviations

This contains all of the grammar rules that are used by the ASPIN parser and semantic analyzer.

Table 6. ASPIN Grammar Rules

<table>
<thead>
<tr>
<th>Rule Name</th>
<th>Production Rule</th>
</tr>
</thead>
<tbody>
<tr>
<td>s1</td>
<td>$s \rightarrow ss$</td>
</tr>
<tr>
<td>s2</td>
<td>$s \rightarrow sp$</td>
</tr>
<tr>
<td>s3</td>
<td>$s \rightarrow sc$</td>
</tr>
<tr>
<td>ss1</td>
<td>$ss \rightarrow d, n \text{ pred}$</td>
</tr>
<tr>
<td>ss2</td>
<td>$ss \rightarrow n \text{ pred}$</td>
</tr>
<tr>
<td>sp1</td>
<td>$sp \rightarrow d, n \text{ pred conj pred}$</td>
</tr>
<tr>
<td>sp2</td>
<td>$sp \rightarrow n \text{ pred conj pred}$</td>
</tr>
<tr>
<td>sc1</td>
<td>$sc \rightarrow ss \text{ conj ss}$</td>
</tr>
<tr>
<td>pred1</td>
<td>$pred \rightarrow avs$</td>
</tr>
<tr>
<td>pred2</td>
<td>$pred \rightarrow avs n$</td>
</tr>
<tr>
<td>pred3</td>
<td>$pred \rightarrow avs n d$</td>
</tr>
<tr>
<td>pred4</td>
<td>$pred \rightarrow avs n d d$</td>
</tr>
<tr>
<td>pred5</td>
<td>$pred \rightarrow avs d$</td>
</tr>
<tr>
<td>pred6</td>
<td>$pred \rightarrow avs d d$</td>
</tr>
<tr>
<td>pred7</td>
<td>$pred \rightarrow pvs$</td>
</tr>
<tr>
<td>pred8</td>
<td>$pred \rightarrow pvs d$</td>
</tr>
<tr>
<td>pred9</td>
<td>$pred \rightarrow pvs d d$</td>
</tr>
<tr>
<td>pred10</td>
<td>$pred \rightarrow pvs d d d$</td>
</tr>
<tr>
<td>pred11</td>
<td>$pred \rightarrow evs n$</td>
</tr>
<tr>
<td>pred12</td>
<td>$pred \rightarrow evs d$</td>
</tr>
<tr>
<td>pred13</td>
<td>$pred \rightarrow evs n d$</td>
</tr>
<tr>
<td>pred14</td>
<td>$pred \rightarrow evs \text{ adj}$</td>
</tr>
<tr>
<td>Rule Name</td>
<td>Production Rule</td>
</tr>
<tr>
<td>-----------</td>
<td>-----------------</td>
</tr>
<tr>
<td>pred15</td>
<td>pred -&gt; evs adjs d</td>
</tr>
<tr>
<td>pred16</td>
<td>pred -&gt; evs adjs n</td>
</tr>
<tr>
<td>avs1</td>
<td>avs -&gt; adv verb</td>
</tr>
<tr>
<td>avs2</td>
<td>avs -&gt; verb</td>
</tr>
<tr>
<td>avs3</td>
<td>avs -&gt; mod vinf</td>
</tr>
<tr>
<td>avs4</td>
<td>avs -&gt; mod adv vinf</td>
</tr>
<tr>
<td>avs5</td>
<td>avs -&gt; mod not vinf</td>
</tr>
<tr>
<td>avs6</td>
<td>avs -&gt; mod not adv vinf</td>
</tr>
<tr>
<td>avs7</td>
<td>avs -&gt; have</td>
</tr>
<tr>
<td>pvs1</td>
<td>pvs -&gt; be ven</td>
</tr>
<tr>
<td>pvs2</td>
<td>pvs -&gt; be not ven</td>
</tr>
<tr>
<td>pvs3</td>
<td>pvs -&gt; be adv ven</td>
</tr>
<tr>
<td>pvs4</td>
<td>pvs -&gt; be not adv ven</td>
</tr>
<tr>
<td>pvs5</td>
<td>pvs -&gt; mod bei ven</td>
</tr>
<tr>
<td>pvs6</td>
<td>pvs -&gt; mod adv bei ven</td>
</tr>
<tr>
<td>pvs7</td>
<td>pvs -&gt; mod not bei ven</td>
</tr>
<tr>
<td>pvs8</td>
<td>pvs -&gt; mod bei adv ven</td>
</tr>
<tr>
<td>pvs9</td>
<td>pvs -&gt; mod not bei adv ven</td>
</tr>
<tr>
<td>evs1</td>
<td>evs -&gt; be</td>
</tr>
<tr>
<td>evs2</td>
<td>evs -&gt; be not</td>
</tr>
<tr>
<td>evs3</td>
<td>evs -&gt; mod not bei</td>
</tr>
<tr>
<td>evs4</td>
<td>evs -&gt; mod adv bei</td>
</tr>
<tr>
<td>d1</td>
<td>d -&gt; prep n</td>
</tr>
<tr>
<td>d2</td>
<td>d -&gt; adv</td>
</tr>
<tr>
<td>d3</td>
<td>d -&gt; scon n</td>
</tr>
<tr>
<td>d4</td>
<td>d -&gt; scon ss</td>
</tr>
<tr>
<td>d5</td>
<td>d -&gt; scon ss conj ss</td>
</tr>
<tr>
<td>n1</td>
<td>n -&gt; np</td>
</tr>
<tr>
<td>n2</td>
<td>n -&gt; np pp</td>
</tr>
<tr>
<td>n3</td>
<td>n -&gt; np rest</td>
</tr>
<tr>
<td>n4</td>
<td>n -&gt; np of n</td>
</tr>
<tr>
<td>n5</td>
<td>n -&gt; np of np pp</td>
</tr>
<tr>
<td>n6</td>
<td>n -&gt; nc</td>
</tr>
<tr>
<td>n7</td>
<td>n -&gt; cl</td>
</tr>
<tr>
<td>n8</td>
<td>n -&gt; cl conj cl</td>
</tr>
<tr>
<td>n9</td>
<td>n -&gt; cl , cl conj cl</td>
</tr>
<tr>
<td>n10</td>
<td>n -&gt; np ( head )</td>
</tr>
<tr>
<td>nc1</td>
<td>nc -&gt; np conj np</td>
</tr>
<tr>
<td>nc2</td>
<td>nc -&gt; np , nc</td>
</tr>
<tr>
<td>np1</td>
<td>np -&gt; predet det adjs head</td>
</tr>
<tr>
<td>Rule Name</td>
<td>Production Rule</td>
</tr>
<tr>
<td>----------</td>
<td>----------------</td>
</tr>
<tr>
<td>np2</td>
<td>np -&gt; predet det head</td>
</tr>
<tr>
<td>np3</td>
<td>np -&gt; predet adjls head</td>
</tr>
<tr>
<td>np4</td>
<td>np -&gt; predet head</td>
</tr>
<tr>
<td>np5</td>
<td>np -&gt; det ord # adjls head</td>
</tr>
<tr>
<td>np6</td>
<td>np -&gt; det ord # head</td>
</tr>
<tr>
<td>np7</td>
<td>np -&gt; det ord adjls head</td>
</tr>
<tr>
<td>np8</td>
<td>np -&gt; det ord head</td>
</tr>
<tr>
<td>np9</td>
<td>np -&gt; det # adjls head</td>
</tr>
<tr>
<td>np10</td>
<td>np -&gt; det # head</td>
</tr>
<tr>
<td>np11</td>
<td>np -&gt; det adjls head</td>
</tr>
<tr>
<td>np12</td>
<td>np -&gt; det head</td>
</tr>
<tr>
<td>np17</td>
<td>np -&gt; # adjls head</td>
</tr>
<tr>
<td>np18</td>
<td>np -&gt; # head</td>
</tr>
<tr>
<td>np19</td>
<td>np -&gt; adjls head</td>
</tr>
<tr>
<td>np20</td>
<td>np -&gt; head</td>
</tr>
<tr>
<td>np22</td>
<td>np -&gt; pron</td>
</tr>
<tr>
<td>head1</td>
<td>head -&gt; noun</td>
</tr>
<tr>
<td>head2</td>
<td>head -&gt; id</td>
</tr>
<tr>
<td>head3</td>
<td>head -&gt; noun head</td>
</tr>
<tr>
<td>head4</td>
<td>head -&gt; id head</td>
</tr>
<tr>
<td>adjls1</td>
<td>adjls -&gt; adj</td>
</tr>
<tr>
<td>adjls2</td>
<td>adjls -&gt; adj , adjls</td>
</tr>
<tr>
<td>adj1</td>
<td>adj -&gt; ving</td>
</tr>
<tr>
<td>adj2</td>
<td>adj -&gt; ven</td>
</tr>
<tr>
<td>pp1</td>
<td>pp -&gt; prep n</td>
</tr>
<tr>
<td>pp2</td>
<td>pp -&gt; prep conj prep n</td>
</tr>
<tr>
<td>rest1</td>
<td>rest -&gt; rcon pred</td>
</tr>
<tr>
<td>rest2</td>
<td>rest -&gt; cl</td>
</tr>
<tr>
<td>cl1</td>
<td>cl -&gt; nvs</td>
</tr>
<tr>
<td>cl2</td>
<td>cl -&gt; nvs d</td>
</tr>
<tr>
<td>cl3</td>
<td>cl -&gt; n nvs</td>
</tr>
<tr>
<td>cl4</td>
<td>cl -&gt; n nvs d</td>
</tr>
<tr>
<td>cl5</td>
<td>cl -&gt; n xvs</td>
</tr>
<tr>
<td>cl6</td>
<td>cl -&gt; xvs</td>
</tr>
<tr>
<td>cl7</td>
<td>cl -&gt; n xvs d</td>
</tr>
<tr>
<td>cl8</td>
<td>cl -&gt; xvs d</td>
</tr>
<tr>
<td>cl9</td>
<td>cl -&gt; n ivs n</td>
</tr>
<tr>
<td>cl10</td>
<td>cl -&gt; n ivs</td>
</tr>
<tr>
<td>cl11</td>
<td>cl -&gt; ivs n</td>
</tr>
<tr>
<td>cl12</td>
<td>cl -&gt; ivs</td>
</tr>
<tr>
<td>Rule Name</td>
<td>Production Rule</td>
</tr>
<tr>
<td>-----------</td>
<td>-----------------</td>
</tr>
<tr>
<td>cl13</td>
<td>cl -&gt; n ivs n d</td>
</tr>
<tr>
<td>cl14</td>
<td>cl -&gt; n ivs d</td>
</tr>
<tr>
<td>cl15</td>
<td>cl -&gt; ivs n d</td>
</tr>
<tr>
<td>cl16</td>
<td>cl -&gt; ivs d</td>
</tr>
<tr>
<td>cl17</td>
<td>cl -&gt; n gvs</td>
</tr>
<tr>
<td>cl18</td>
<td>cl -&gt; gvs n</td>
</tr>
<tr>
<td>cl19</td>
<td>cl -&gt; gvs</td>
</tr>
<tr>
<td>cl20</td>
<td>cl -&gt; gvs n d</td>
</tr>
<tr>
<td>cl21</td>
<td>cl -&gt; gvs d</td>
</tr>
<tr>
<td>nvs1</td>
<td>nvs -&gt; ven</td>
</tr>
<tr>
<td>xvs1</td>
<td>xvs -&gt; to bei ven</td>
</tr>
<tr>
<td>ivs1</td>
<td>ivs -&gt; to vinf</td>
</tr>
<tr>
<td>gvs1</td>
<td>gvs -&gt; ving</td>
</tr>
</tbody>
</table>

Below are definitions for the grammar abbreviations used in this work.

- **s**: sentence
- **ss**: simple sentence
- **sp**: sentence with a compound predicate
- **sc**: compound sentence
- **pred**: predicate
- **avs**: active verb sequence
- **pvs**: passive verb sequence
- **evs**: attributive verb sequence
- **d**: adverbial
- **n**: nominal
- **nc**: compound nominal structure
- **np**: noun phrase
- **head**: head
- **adjs**: adjective sequence
- **adj**: adjective
- **pp**: prepositional phrase
- **rest**: restrictive relative clause
- **cl**: verbal phrase or clause
- **nvs**: past participle verb sequence
- **xvs**: future perfect verb sequence
- **ivs**: infinitive verb sequence
- **gvs**: present participle verb sequence
conj: conjunction
verb: verb
mod: modal
not: not
adv: simple adverb
have: form of the verb have
be: is form of the verb be
bei: be form of the verb be
prep: preposition
scon: subordinate conjunction
of: the preposition of
predet: determiner such as all, none
det: determiner
#: cardinal number
ord: ordinal such as first, last
pron: pronoun
rcon: restrictive relative clause conjunction
ven: past participle form of a verb
vinf: infinitive form of a verb
ving: present participle form of a verb.
Appendix C. Prolog Source Code

This appendix includes the semantic analyzer Prolog source code. The semantic analyzer is made up of four files, plus a file that contains the input parse trees to be analyzed. Included in this appendix are three of the files: SEMRULES.PL, MAIN.PL, and CANON.PL. Semrules.pl contains the semantic rules written for each grammatical structure. Main.pl contains the analysis engine as well as the select and attach predicates used in joining conceptual graphs. Canon.pl contains the conceptual type hierarchy and the canonical graphs for the words currently covered by the vocabulary of the semantic analyzer. The file ROOTS.PL was omitted for space considerations. It contains facts which are used to determine the root word forms of the words in the vocabulary.

%*********************************************************************
%*                                                                    
%* SEMRULES.PL                                                      
%*                                                                    
%* This file, SEMRULES.PL, contains the semantic rules used by the  
%* semantic analyzer to guide the attachment of canonical graphs. The 
%* rules are based on the grammatical structures used to parse the  
%* sentences.                                                      
%*                                                                    
%*********************************************************************

% %
% adj1 => ving
% ***** This rule is tested by suite sentence #53 *****
% This predicate finds the root of present participle VING and fetches its canonical graph.
% The nominal modified is attached to the canonical of the VING root using the structural role
% marker agnt. The resulting graph is added to the relative graph list. The modified nominal is
% passed back for further attachments.
% p(adj1,[t(ving,Part)],Gi,Gi) :- nl,write(adj1),
% root(Part,Root),
% g(T,Root,L,R,Conf,Gindex),
% attach(agnt,_,Gi,g(T,Root,L,R,Conf,Gindex),G),
% retract(relative(List)),
% check_rel(List,G,K),
% asserta(relative(K)),
% write([adj1]), nl.
%
%
% adj2 => ven
% ***** This rule is tested by suite sentence #68 *****
% This predicate finds the root of past participle VEN and fetches its canonical graph. The
% nominal modified is attached to the canonical of the VEN root using the structural role marker
% obj. The resulting graph is added to the relative graph list. The modified nominal is passed
% back for further attachments.
% p(adj2,[t(ven,Ven)],Gi,Gi) :- nl,write(adj2),
% root(Ven,Root),
% g(T,Root,L,R,Conf,Gindex),
% attach(obj,_,Gi,g(T,Root,L,R,Conf,Gindex),G),
% retract(relative(List)),
% check_rel(List,G,K),
% asserta(relative(K)),
% write([adj2]), nl.
%
%
% adj1a => adj
% ***** This rule is tested by suite sentence #53 *****
% This predicate assumes that the adjective making up the adjective sequence is of the form of a
% adj1 or adj2 structure. Therefore, this rules passes the graph of the modified nominal Gi, for
% further processing by the other adjective rules.
% p(adj1,[c(Ra,La)],Gi,G) :- nl, write(adj1a),
% p(Ra,La,Gi,G),
% write([adj1a]), nl.
%
%
% adj1b => adj
% ***** This rule is tested by suite sentence #56 *****
% This version of adj1 handles the case in which the adjective sequence is made up of a pure
% adjective. This rule looks for a canonical graph of the adjective and attaches it to the graph of the
% modified nominal according to the relationship given in the graph of the adjective. The graph of
% the adjective contains a list of types to which the adjective can be attached. The adjective also
% contains the name of role which it plays. Note that the cut, !, is necessary for the cases where
% there are multiple canonical graphs for the words in a sentence. Without the cut, the analyzer is
% likely to backtrack and try the next adj rule, which cause a loss of information, since this rule
attaches the adjective using a specific relation.
\[
p(\text{adjs1}, [(\text{adj}, \text{Adj})], g(\text{Tin}, \text{Nin}, \text{Lin}, \text{Rin}, \text{Confin}, \text{Gxin}), g(\text{Tin}, \text{Nin}, \text{Jin}, \text{Rout}, \text{Confin}, \text{Gxin})):~
\]
\nwrite(adjs1b),
\n(\text{Type}, \text{Adj}, \text{L}, \text{R}, \text{Conf}, \text{Gindx}),
\n\text{attach}([g(\text{Gindx}, \text{Gindx}, [], g(\text{Type}, \text{Adj}, \text{L}, \text{R}, \text{Conf}, \text{Gindx}, g(\text{Ta}, \text{Adj}, \text{La}, \text{Ra}, \text{Ca}, \text{Gxa})),
\n\text{select}(\text{adj}, \text{Role}, \text{Tin}, g(\text{Ta}, \text{Adj}, [], \text{Ra}, \text{Ca}, \text{Gxa}), \text{La}, \text{Slot}), !,
\nappend([\text{Rin}, [\text{Slot}], \text{Rout}]),
\nwrite([\text{adjs1b}]), \text{nl}.
\]
\%
%
%
\%
\adjs2a \Rightarrow \text{adj adjs}
%
\%

***** This rule is tested by suite sentence #68 *****
%

This predicate assumes that the first adjective making up the adjective sequence is of the form of
%
a \text{adj1} or \text{adj2} structure. Therefore, this rules passes the graph of the modified nominal, \text{Gi}, for
%
further processing by the other adjective rules. The resulting graph \text{Gx} is then passed for the
%
processing of the \text{adjs} structure.
\%
\%
\%
\adjs2b \Rightarrow \text{adj adjs}
%
\%

***** This rule is tested by suite sentence #69 *****
%

This version, \text{adjs2b}, finds the graph for the adjective and attaches it to the modified nominal in
%
a similar fashion to \text{adjs1b}. After the adj is attached, the resulting graph is passed to the \text{adjs}
%
structure for processing. Again, the cut is necessary to prevent a loss of information.
\%
\%
\%
\adjs3a \Rightarrow \text{adj , adjs}
%
\%

***** This rule is tested by suite sentence #70 *****
%

This predicate assumes that the first adjective making up the adjective sequence is of the form of
%
a \text{adj1} or \text{adj2} structure. Therefore, this rules passes the graph of the modified nominal, \text{Gi}, for
%
further processing by the other adjective rules. The resulting graph \text{Gx} is then passed for the
%
processing of the \text{adjs} structure.
\%
\%
\%
\adjs3b \Rightarrow \text{adj , adjs}
%

***** This rule is tested by suite sentence #67 *****
This version, adj3b, finds the graph for the adjective and attaches it to the modified nominal in a similar fashion to adj3b. After the adj is attached, the resulting graph is passed to the adj structure for processing. Again, the cut is necessary to prevent a loss of information.

\[p(adj3b, [t(adj, Adj), l(_-), c(Rb, Lb)], g(Tin, Nin, Lin, R, Confin, Gxin), G)]::

\[nl, write(adj3b),
\]
\[g(Type, Adj, L, R, Conf, Gindx),
\]
\[attach([\{gindx, Gindx, []\}], g(Type, Adj, L, R, Conf, Gindx), g(Ta, Adj, La, Ra, Ca, Gxa)),
\]
\[select(adj, Role, Tin, g(Ta, Adj, [], Ra, Ca, Gxa), La, Slot),
\]
\[append(Rin, [Slot], Rout),
\]
\[p(Rb, Lb, g(Tin, Nin, Lin, Rout, Confin, Gxin), G),
\]
\[write([adj3b]), nl.\]

\[avs1 \Rightarrow \text{adv verb}\]

\[***** This rule is tested by suite sentence #1 *****\]

\[Predicate AVS1 first finds the root of the verb form. The graph for the root is then retrieved from the canon. The canonical graph for the adverb is then retrieved and attached to the graph of the verb. The resulting graph is then passed back for later attachments.\]

\[p(avs1, [t(adv, Adv), t(verb, Verb)], G) : nl, write(avs1),
\]
\[root(V, Root), g(T, Root, CL, RL, Conf, Gindx),
\]
\[attach([\{gindx, Gindx, []\}], g(T, Root, CL, RL, Conf, Gindx), Ga),
\]
\[g(Ta, Adv, Sa, Fa, Confa, Gindxa),
\]
\[Gadv = g(Ta, Adv, Sa, Fa, Confa, Gindxa),
\]
\[attach(adv, _, Gadv, Ga, G),
\]
\[write([avs1]), nl.\]

\[avs2 \Rightarrow \text{verb}.\]

\[***** This rule is tested by suite sentence #57 *****\]

\[Predicate AVS2 retrieves the canonical graph for the root form of the verb and passes it back for later attachments.\]

\[p(avs2, [t(verb, V)], Ga) : nl, write(avs2),
\]
\[root(V, Root),
\]
\[g(T, Root, CL, RL, Conf, Gindx),
\]
\[attach([\{gindx, Gindx, []\}], g(T, Root, CL, RL, Conf, Gindx), Ga),
\]
\[write([avs2]), nl.\]

\[avs3 \Rightarrow \text{mod vinf}\]

\[***** This rule is tested by suite sentence #2 *****\]

\[Predicate AVS3 retrieves the canonical graph for the root form of the verb, to which the monadic conceptual relation 'modal' is attached with a referent represented by the variable Mod. The resulting graph is then passed back for further processing.\]

\[p(avs3, [t(mod, Mod), t(vinf, Vinf)], G) : nl, write(avs3),
\]
\[root(Vinf, Root), g(T, Root, CL, RL, Conf, Gindx),
\]
\[attach([\{gindx, Gindx, []\}], g(T, Root, CL, RL, Conf, Gindx), Ga),
\]
\[attach([\{mod, Mod, []\}], Ga, G),
\]
\[write([avs3]), nl.\]

\[avs4 \Rightarrow \text{mod adv vinf}\]
***** This rule is tested by suite sentence #3 *****
Predicate AVS4 retrieves the canonical graph for the root form of the verb. The graph of the
adverb is then retrieved and attached to the graph of the verb. The modal is then attached to the
graph of the verb using the monadic conceptual relation 'modal.' The resulting graph is then
passed back for further processing.
p(avs4,[t(mod,Mod),t(adv,Adv),t(vinf,Vinf)],G) :- nl, write(avs4),
root(Vinf,Root), g(T,Root,CL,RL,Conf,Gindx),
attach([[gindx,Gindx,[]],g(T,Root,CL,RL,Conf,Gindx),Gx]),
g(Ta,Adv,Sa,Fa,Conf,Fa,Gindx),
attach(adv,_,g(Ta,Adv,Sa,Fa,Conf,Fa,Gindx),Gx,Ga),
attach([[mod,Mod,[]],Ga,G]),
write([avs4]), nl.

av5 = mod not vinf

***** This rule is tested by suite sentence #4 *****
Predicate AVS5 retrieves the canonical graph for the root form of the verb. The monadic
relation 'negation' is added using the contents of the variable 'Not' as the referent. The
monadic relation 'modal' is also added to the graph of the verb using the contents of the
variable 'Mod' as the referent. The resulting graph is then passed back for further processing.
p(avs5,[t(mod,Mod),t(not,Not),t(vinf,Vinf)],G) :- nl, write(avs5),
root(Vinf,Root), g(T,Root,CL,RL,Conf,Gindx),
attach([[gindx,Gindx,[]],g(T,Root,CL,RL,Conf,Gindx),Gx]),
attach([[neg,Not,[]],Gx,Ga]),
attach([[mod,Mod,[]],Ga,G]),
write([avs5]), nl.

av6 = mod not adv vinf

***** This rule is tested by suite sentence #5 *****
Predicate AVS6 retrieves the canonical graph for the root form of the verb. The monadic
relation 'negation' is added using the contents of the variable 'Not' as the referent. The
monadic relation 'modal' is also added to the graph of the verb using the contents of the
variable 'Mod' as the referent. The canonical graph for the adverb is then retrieved and
attached to the graph of the verb. The resulting graph is then passed back for further processing.
p(avs6,[t(mod,Mod),t(not,Not),t(adv,Adv),t(vinf,Vinf)],G) :- nl, write(avs6),
root(Vinf,Root), g(T,Root,CL,RL,Conf,Gindx),
attach([[gindx,Gindx,[]],g(T,Root,CL,RL,Conf,Gindx),Gx]),
attach([[neg,Not,[]],Gx,Ga]),
attach([[mod,Mod,[]],Ga,Gb]),
g(Ta,Adv,Sa,Fa,Conf,Fa,Gindx),
attach(adv,_,g(Ta,Adv,Sa,Fa,Conf,Fa,Gindx),Gb,G),
write([avs6]), nl.

av7 = have

***** This rule is tested by suite sentence #6 *****
Predicate AVS7 retrieves the canonical graph for the root form of the 'have' type verb
encountered in the sentence. The graph is then passed back for further processing.
p(avs7,[t(have,Have)],G) :- nl, write(avs7),
root(Have,Root), g(T,Root,CL,RL,Conf,Gindx),
attach([([gindx,Gindx,[]]),g(T,Root,Cl,RI,Conf,Gindx),Gx),
write([av57]),nl.

% %
c11 => nvs
%%%% This rule is tested by suite sentence #32 %%%%
% Predicate CL1 processes the nvs structure and passes the resulting graph back for
% further processing.
p(c11,[c(Ra,La)],G) :- nl, write(c11),
p(Ra,La,G),
write([c11]),nl.

% %
c12 => nvs d
%%%% This rule is tested by suite sentence #50 %%%%
% Predicate CL2 first processes the nvs structure and then the adverbial structure. The graph
% of the adverbial is then attached to the graph of the verb in the nvs structure based on the
% preposition 'Prep.' The resulting graph is then passed back for further processing.
p(c12,[c(Ra,La),c(Rd,Ld)],G) :- nl, write(c12),
p(Ra,La,Gv),
p(Rd,Prep,Ld,Gd),
attach(Prep,_,Gd,Gv,G),
write([c12]),nl.

% %
c13 => n nvs
%%%% This rule is tested by suite sentence #41 %%%%
% Predicate CL3 first processes the nominal structure and then the nvs structure. The graph of the
% nominal is then attached to the graph of the verb in the nvs structure. The resulting graph is then
% passed back for further processing.
p(c13,[c(Ra,La),c(Rb,Lb)],G) :- nl, write(c13),
p(Ra,La,Gn),
p(Rb,Lb,Gv),
attach(obj,_,Gn,Gv,G),
write([c13]),nl.

% %
c14 => n nvs d
%%%% This rule is tested by suite sentence #51 %%%%
% Predicate CL4 processes the nominal structure followed by the nvs and adverbial structures. The
% nominal graph is then attached to the nvs graph using the structural role marker 'agt.' The
% graph of the adverbial is then attached to the nvs graph based on the preposition 'Prep.' The
% resulting graph is attached to the relative graph list. The nominal graph is then passed back for
% further processing.
p(c14,[c(Ra,La),c(Rh,Lh),c(Rc,Lc)],Gn) :- nl, write(c14),
p(Ra,La,Gn),
p(Rh,Lh,Gv),
p(Rc,Prep,Lc,Gd),
attach(obj,_,Gn,Gv,Gt),
attach(Prep,_,Gd,Gt,Gt),
retract(relative(L)),check_rel(L,G,K),
asserta(relative(K)),
write([cl4]), nl.

% cl5 => n xvs
% ***** This rule is tested by suite sentence #10 *****
% Predicate CL5 processes the nominal structure followed by the xvs structure. The graph of the
% nominal is attached to the graph of the verb in the xvs structure using the 'obj' structural role
% marker. The resulting graph is then attached to the relative graph list. The nominal graph is
% passed back for further processing.
% p(cl5,[c(Rn,Ln),c(Rx,Lx)],Gn) :- nl, write(cl5),
% p(Rn,Ln,Gn),
% p(Rx,Lx,Gx),
% attach(obj,_,Gn,Gx,G),
% retract(relative(L)),check_rel(L,G,K),
% asserta(relative(K)),
% write([cl5]), nl.

% cl6 => xvs
% ***** This rule is tested by suite sentence #58 *****
% Predicate CL6 processes the xvs structure and passes the resulting graph for further processing.
% p(cl6,[c(Ra,La)],G) :- nl, write(cl6),
% p(Ra,La,G),
% write([cl6]), nl.

% cl7 => n xvs d
% ***** This rule is tested by suite sentence #11 *****
% Predicate CL7 processes the nominal, xvs, and adverbial structures. The graph of the nominal is
% attached to the verb of the xvs graph using the relation obj. The adverbial graph is then attached
% to the verb graph based on the preposition which acts of a literal role marker. The resulting
% graph is then attached to the relative graph list. The nominal graph is passed back for further
% processing.
% p(cl7,[c(Rn,Ln),c(Rx,Lx),c(Rd,Ld)],Gn) :- nl, write(cl7),
% p(Rn,Ln,Gn),
% p(Rx,Lx,Gx),
% p(Rd,Prep,Ld,Gd),
% attach(obj,_,Gn,Gx,Ga),
% attach(Prep,_,Gd,Ga,G),
% retract(relative(L)),check_rel(L,G,K),
% asserta(relative(K)),
% write([cl7]), nl.

% cl8 => xvs d
% ***** This rule is tested by suite sentence #42 *****
% Predicate CL8 processes the xvs and adverbial structures. The graph of the adverbial is attached
% to the graph of the verb in the xvs structure. The attachment is based on the preposition, which
% acts as a literal role marker. The resulting graph is passed back for further processing.
% p(cl8,[c(Rx,Lx),c(Rd,Ld)],G) :- nl, write(cl8),

Appendix C. Prolog Source Code
p(Rx,Lx,Gx),
p(Rd,Prep,Ld,Gd),
attach(Prep,_Gd,Gx,G),
write([cl8]), nl.
%
%
cl9 => n ivs n
% ***** This rule is tested by suite sentence #43 *****
% Predicate CL9 processes the nominal structure followed by the ivs structure and second nominal
% structure. A copy of the first nominal graph is attached to the verb of the infinitive verb
% sequence using the structural role marker 'agnt.' The second nominal graph is attached to the
% graph using the structural role marker 'obj'. This graph is added to the relative graph list.
% The graph of the first nominal is passed back for further processing.
pl(cl9,[c(Rn1,Ln1),c(Ri,li),c(Rn2,Ln2)],Gn1) :- nl, write(cl9),
p(Rn1,Ln1,Gn1),
p(Ri,li,Gi),
p(Rn2,Ln2,Gn2),
attach(agnt,_,Gn1,Gi,Ga),
attach(obj,_,Gn2,Ga,G),
retract(relative(Li),check_rel(L,G,K)),
asserta(relative(K)),
write([cl9]), nl.
%
%
cl10 => n ivs
% ***** This rule is tested by suite sentence #44 *****
% Predicate CL10 processes the nominal structure followed by the ivs structure. A copy of the
% nominal graph is attached to the graph of the verb. The resulting graph is added to the relative
% graph list. The graph of the nominal is then passed back for further processing.
pl(cl10,[c(Rn,Ln),c(Ri,li)],Gn) :- nl, write(cl10),
p(Rn,Ln,Gn),
p(Ri,li,Gi),
attach(obj,_,Gn,Gi,G),
retract(relative(Li),check_rel(L,G,K)),
asserta(relative(K)),
write([cl10]), nl.
%
%
cl11 => ivs n
% ***** This rule is tested by suite sentence #45 *****
% Predicate CL11 processes the ivs structure followed by the nominal structure. The graph of the
% nominal is attached to the graph of the verb in the ivs structure using the structural role marker
% 'obj.' The resulting graph is passed back for further processing.
pl(cl11,[c(Ri,li),c(Rn,Ln)],G) :- nl, write(cl11),
p(Ri,li,Gi),
p(Rn,Ln,Gn),
attach(obj,_,Gn,Gi,G),
write([cl11]), nl.
%
%
cl12 => ivs
% ***** This rule is tested by suite sentence #16 *****
% Predicate CL12 processes the ivs structure and passes the resulting graph back for further
% attachments.
p(cl12,[c(Ri,Li)],G) :- nl, write(cl12),
p(Ri,Li,G),
write([cl12]), nl.
%
%
c1l3 => n ivs n d
% ***** This rule is tested by suite sentence #46 *****
% Predicate CL13 processes the nomina structure, ivs structure, second nominal structure, and
% adverbial structure. A copy of the first nominal structure is attached to the graph of the verb in
% the ivs structure using the structural role marker 'agt.' The graph of the second nominal
% structure is attached to the graph of the verb using the structural role marker 'obj.' The graph of
% the adverbial structure is then attached to the verb using the preposition as a literal role marker.
% The resulting graph is added to the relative graph list. The graph of the first nominal is passed
% back for further processing.
p(cl13,[c(Rn1,Ln1),c(Rn2,Ln2),c(Rd,Ld)],Gn1) :- nl, write(cl13),
p(Rn1,Ln1,Gn1),
p(Ri,Li,Gi),
p(Rn2,Ln2,Gn2),
p(Rd,Prep,Ld,Gd),
attach(agt,_,Gn1,Gi,Ga),
attach(obj,_,Gn2,Ga,Gb),
attach(Prep,_,Gd,Gb,G),
retract(relative(L)),check_rel(L,G,K),
asserta(relative(K)),
write([cl13]), nl.
%
%
c1l4 => n ivs n d
% ***** This rule is tested by suite sentence #47 *****
% Predicate CL14 processes the nominal structure, the ivs structure, and the adverbial structure.
% The graph of the nominal is then attached to the graph of the verb of the ivs structure using the
% structural role marker 'obj.' The graph of the adverbial is then attached to the graph of the verb
% of the ivs structure using the preposition as a literal role marker. The resulting graph is added to
% the relative graph list. The graph of the nominal is passed back for further processing.
p(cl14,[c(Rn,Ln),c(Ri,Li),c(Rd,Ld)],Gn) :- nl, write(cl14),
p(Rn,Ln,Gn),
p(Ri,Li,Gi),
p(Rd,Prep,Ld,Gd),
attach(obj,_,Gn,Gi,Ga),
attach(Prep,_,Gd,Gb,G),
retract(relative(L)),check_rel(L,G,K),
asserta(relative(K)),
write([cl14]), nl.
%
%
c1l5 => ivs n d
% ***** This rule is tested by suite sentence #48 *****
% Predicate CL15 processes the ivs structure, nominal structure, and the adverbial structure. The
% nominal graph is then attached to the graph of the verb of the ivs structure using the structural role marker 'obj.' The adverbial graph is then attached to the verb based on the preposition, which is a literal role marker. The resulting graph is passed back for further processing.

\[
\text{p(c115, [c(Ri, Li), c(Rn, Ln), c(Rd, Ld)], G) : - nl, write(c115),}
\text{p(Ri, Li, Gi),}
\text{p(Rn, Ln, Gn),}
\text{p(Rd, Prep, Ld, Gd),}
\text{attach(obj, __, Gn, Gi, Gb),}
\text{attach(Prep, __, Gd, Gb, G),}
\text{write([c115]), nl.}
\]

% %
% c116 => ivs d
% ***** This rule is tested by suite sentence #49 *****
% Predicate CL16 processes the ivs structure then processes the adverbial structure. The graph of the adverbial structure is then attached to the graph of the verb of the ivs structure using the preposition as a literal role marker. The resulting graph is passed back for further processing.

\[
\text{p(c116, [c(Ri, Li), c(Rd, Ld)], G) : - nl, write(c116),}
\text{p(Ri, Li, Gi),}
\text{p(Rd, Prep, Ld, Gd),}
\text{attach(Prep, __, Gd, Gi, G),}
\text{write([c116]), nl.}
\]

% %
% c117 => n gvs
% ***** This rule is tested by suite sentence #59 *****
% Predicate CL17 processes the nominal structure and the gvs structure. The nominal graph is attached to the verb of the gvs structure using the structural role marker 'agt.' The resulting graph is then passed back for further processing.

\[
\text{p(c117, [c(Rn, Ln), c(Rg, Lg)], G) : - nl, write(c117),}
\text{p(Rn, Ln, Gn),}
\text{p(Rg, Lg, Gg),}
\text{attach(agt, __, Gn, Gg, G),}
\text{write([c117]), nl.}
\]

% %
% c118 => gvs n
% ***** This rule is tested by suite sentence #59 *****
% Predicate CL18 processes the gvs structure and the nominal structure. The graph of the nominal structure is then attached to the graph of the verb of the gvs structure using the structural role marker 'obj.' The resulting graph is then passed back for later attachments.

\[
\text{p(c118, [c(Rg, Lg), c(Rn, Ln)], G) : - nl, write(c118),}
\text{p(Rg, Lg, Gg),}
\text{p(Rn, Ln, Gn),}
\text{attach(obj, __, Gn, Gg, G),}
\text{write([c118]), nl.}
\]

% %
% c119 => gvs
% ***** This rule is tested by suite sentence #7 *****
% Predicate CL19 processes the gvs structure and passes the resulting graph back for further
nominal graph is then attached to the graph of the verb of the ivs structure using the structural
role marker 'obj'. The adverbial graph is then attached to the verb based on the preposition,
which is a literal role marker. The resulting graph is passed back for further processing.

\begin{verbatim}
\text{o}(\text{cl15}, [\text{c(Rn,Ln)}, \text{c(Rn,Ln)}, \text{c(Rn,Ld)}], G) :- nl, write(cl15),
p(Rn,Ln,Gn),
p(Rd,Prep,Ld,Gd),
attach(obj,...,Gn,Gi,Gb),
attach(Prep,...,Gd,Gi,Gb),
write([cl15]), nl.
\end{verbatim}

\text{cl16} :- ivs d

\text{**** This rule is tested by suite sentence \#49 ****}

Predicate CL16 processes the ivs structure then processes the adverbial structure. The graph of
the adverbial structure is then attached to the graph of the verb of the ivs structure using the
preposition as a literal role marker. The resulting graph is passed back for further processing.

\begin{verbatim}
\text{o}(\text{cl16}, [\text{c(Rn,Li)}, \text{c(Rn,Ld)}], G) :- nl, write(cl16),
p(Ri,Li,Gi),
p(Rd,Prep,Ld,Gd),
attach(Prep,...,Gd,Gi,Gb),
write([cl16]), nl.
\end{verbatim}

\text{cl17} :- n gvs

\text{**** This rule is tested by suite sentence \#59 ****}

Predicate CL17 processes the nominal structure and the gvs structure. The nominal graph is
attached to the verb of the gvs structure using the structural role marker 'agt.' The resulting
graph is then passed back for further processing.

\begin{verbatim}
\text{o}(\text{cl17}, [\text{c(Rn,Ln)}, \text{c(Rn,Lg)}], G) :- nl, write(cl17),
p(Rn,Ln,Gn),
p(Rg,Lg,Gg),
attach(agt,...,Gn,Gg,Gb),
write([cl17]), nl.
\end{verbatim}

\text{cl18} :- gvs n

\text{**** This rule is tested by suite sentence \#59 ****}

Predicate CL18 processes the gvs structure and the nominal structure. The graph of the nominal
structure is then attached to the graph of the verb of the gvs structure using the structural role
marker 'obj.' The resulting graph is then passed back for later attachments.

\begin{verbatim}
\text{o}(\text{cl18}, [\text{c(Rn,Lg)}, \text{c(Rn,Ln)}], G) :- nl, write(cl18),
p(Rg,Lg,Gg),
p(Rn,Ln,Gn),
attach(obj,...,Gn,Gg,Gb),
write([cl18]), nl.
\end{verbatim}

\text{cl19} :- gvs

\text{**** This rule is tested by suite sentence \#7 ****}

Predicate CL19 processes the gvs structure and passes the resulting graph back for further
processing.
p(cl19,[c(Rg,Lg)],G) :- nl, write(cl19),
p(Rg,Lg,G).
write((cl19)), nl.

cl20 => gvs nd

***** This rule is tested by suite sentence #9 *****
Predicate CL20 processes the gvs structure, the nominal structure, and the adverbial structure.
The graph of the nominal structure is attached to the graph of the verb in the gvs structure using the
structural role marker 'obj.' The graph of the adverbial is then attached to the graph of the
verb using the preposition of the adverbial as a literal role marker. The resulting graph is then
passed back for further processing.
p(cl20,[c(Rg,Lg),c(Rn,Ln),c(Rd,Ld)],G) :- nl, write(cl20),
p(Rg,Lg,Gg),
p(Rn,Ln,Gn),
p(Rd,Prep,Ld,Gd),
attach(obj,_,Gn,Gg,Ga),
attach(Prep,_,Gd,Ga,G),
write((cl20)), nl.

c121 => gvs d

***** This rule is tested by suite sentence #8 *****
Predicate CL21 processes the gvs structure followed by the adverbial structure. The graph of the
adverbial is then attached to the graph of the verb of the gvs structure using the preposition of the
adverbial as a literal role marker.
p(cl21,[c(Rg,Lg),c(Rd,Ld)],G) :- nl, write(cl21),
p(Rg,Lg,Gg),
p(Rd,Prep,Ld,Gd),
attach(Prep,_,Gd,Gg,G),
write((cl21)), nl.

d1a => prep n

***** This rule is tested by suite sentence #55 *****
The predicate D1A handles the case of the adverbial 'as a result.' The nominal structure is
processed and is passed back. The preposition 'as a result' is passed back and used by one of the
'select' predicates called by the attach rules in the file 'main.pl.'
p(d1,'as a result',[t(Prep,as),c(Ra,La)],G) :- nl, write(d1a),
p(Ra,La,G),
write((d1a)), nl.

d1b => prep n

***** This rule is tested by suite sentence #54 *****
The predicate D1B processes the nominal and passes it back with the preposition for further
processing.
p(d1,Prep,[t(Prep,Prep),c(Ra,La)],G) :- nl, write(d1b),
p(Ra,La,G),
write((d1b)), nl.
d2 => adv

**** This rule is tested by suite sentence #19 ****
Predicate D2 hands the case where the adverbiale structure is and adverb. The canonical graph of
the adverb is retrieved from the canon and passed back for further processig. Since there is no
prepostion to pass back, the marker 'adv' is used. The 'adv' marker will trigger the use of a
special attach rule in 'main.pl'.
p(d2,adv,[t(adv,Adv),\G]) : nl, write(d2),
g(Ta,Adv,Sa,Fa,Conf,Gindx),
G=g(Ta,Adv,Sa,Fa,Conf,Gindx),
write([d2]), nl.


d3 => scon n

**** This rule is tested by suite sentence #53 ****
Predicate D3 processes the nominal structure and passes the resulting graph back for further
processing. The contents of the variable 'scon' is passed back and used as a literal role marker
during the attach operation.
p(d3,scon,[t(scon,Scon),c(Rn,Ln)],\G) : nl, write(d3),
p(Rn,Ln,G),
write([d3]), nl.


d4 => scon ss

**** This rule is tested by suite sentence #57 ****
Predicate D4 processes the ss structure. The resulting graph is then passed back for further
processing. The contents of the variable 'scon' is passed back and used as a literal role marker
during the attach operation.
p(d4,scon,[t(scon,Scon),c(Ra,La)],\G) : nl, write(d4),
p(Ra,La,G),
write([d4]), nl.


d5 => scon ss conj ss

**** This rule is tested by suite sentence #65 ****
Predicate D5 processes the two ss structures. The graphs for the two ss structures are attached to
a conjunction concept. The type of the conjunction concept is the minimal common supertype
of the two ss graphs. This conjunction concept is then passed back for further processing. The
contents of the variable 'scon' is also passed back and used as a literal role marker which is used
during the attach operation.
p(d5,Prep,[t(scon,Scon),c(Rss1,Lss1),t(conj,Conj),c(Rss2,Lss2)],\G) : nl, write(d5),
p(Rss1,Lss1,Gs1),
g(Ts1,...,...,...)=Gs1,
p(Rss2,Lss2,Gs2),
g(Ts2,...,...,...)=Gs2,
get_ccm_sup(Ts1,Ts2,Conj),
G=g(Conj,Conj,\[],[\(Conj,null,Gs1\),\(Conj,null,Gs2\)],1,1),
write([d5]), nl.
% evs1 => be
% ***** This rule is tested by suite sentence #58 *****
% Predicate EVS1 retrieves the canonical graph for the contents of the variable 'Be.' This graph is
% then passed back for further processing.
p(evs1,[t(be,Be)],Ga) :- nl, write(evs1),
g(T,Be,CL,RL,Conf,Gindx),
attach([[(gindx,Gindx,:[])],g(T,Be,CL,RL,Conf,Gindx),Ga]),
write([evs1]),nl.

% evs2 => be not
% ***** This rule is tested by suite sentence #12 *****
% Predicate EVS2 retrieves the canonical graph for the contents of the variable 'Be.' The monadic
% conceptual relation 'negation' is then attached to the graph. The resulting graph is then passed
% back for further processing.
p(evs2,[t(be,Be),t(not,Not)],G) :- nl, write(evs2),
g(T,Be,CL,RL,Conf,Gindx),
attach([[(gindx,Gindx,:[])],g(T,Be,CL,RL,Conf,Gindx),Gx]),
attach([[(neg,Not,:[])],Gx,G]),
write([evs2]),nl.

% evs3 => mod not bei
% ***** This rule is tested by suite sentence #13 *****
% Predicate EVS3 finds the root of the contents of the variable 'Bei.' The canonical graph for the
% root is retrieved and the monadic conceptual relations 'negation' and 'modal' are attached to it.
% The resulting graph is then passed back for further processing.
p(evs3,[t(mod,Mod),t(not,Not),t(bei,Bei)],G) :- nl, write(evs3),
root(bei,Root),
g(T,Root,CL,RL,Conf,Gindx),
attach([[(gindx,Gindx,:[])],g(T,Root,CL,RL,Conf,Gindx),Gx]),
attach([[(neg,Not,:[])],Gx,Ga]),
attach([[(mod,Mod,:[])],Ga,G]),
write([evs3]),nl.

% evs4 => mod adv bei
% ***** This rule is tested by suite sentence #14 *****
% Predicate EVS4 finds the root of the contents of the variable 'Bei.' The canonical graph for the
% root is retrieved and the monadic conceptual relation 'modal' is attached to it. Next, the
% canonical graph of the adverb is retrieved and attached to the graph verb contained in 'Bei.' The
% resulting graph is then passed back for further processing.
p(evs4,[t(mod,Mod),t(adv,Adv),t(bei,Bei)],G) :- nl, write(evs4),
root(bei,Root),
g(T,Root,CL,RL,Conf,Gindx),
attach([[(gindx,Gindx,:[])],g(T,Root,CL,RL,Conf,Gindx),Gx]),
attach([[(mod,Mod,:[])],Gx,Ga]),
g(Ta,Adv,Sa,Fa,ConfGa,Gindx),
Gadv=g(Ta,Adv,Sa,Fa,ConfGa,Gindx),
attach(adv,_,Gadv,Ga,G),
% gvs1 => ving
% ***** This rule is tested by suite sentence #59 *****
% Predicate GVS1 finds the root of the verb form contained in 'Ving.' The canonical graph for this
% root is retrieved and passed back for further processing.
p(gvs1, [t(ving,Ving)], g(T, Root, L, R, Conf, Gindex)) :- nl, write(gvs1),
root(Ving, Root).
g(T, Root, L, R, Conf, Gindex),
write([gvs1]), nl.
%
% head1 => noun
% ***** This rule is tested by suite sentence #53 *****
% Predicate HEAD1 finds the root of the noun and retrieves its canonical graph. The canonical
% graph is then passed back for further processing.
p(head1, [t(noun, Noun)], Ga) :- nl, write(head1),
root(Noun, Root),
g(Typ, Root, L, R, Conf, Gindex),
attach([([gindex, Gindex, [],]), g(Typ, Root, L, R, Conf, Gindex), Ga],
write([head1]), nl.
%
% head2 => id
% ***** This rule is tested by suite sentence #57 *****
% Predicate HEAD2 creates a graph for the identifier and passes it back for further processing.
p(head2, [t(id, Id)], G) :- nl, write(head2),
G = g(id, Id, [], [], 1, 1),
write([head2]), nl.
%
% head3a => noun head
% ***** This rule is tested by suite sentence #53 *****
% Predicate HEAD3A processes the head structure and retrieves the canonical graph for the root of
% the noun. If the noun is a subtype of OBJECT, then the graph of the noun is attached to the
% graph of the head structure using the conceptual relation 'in.' The resulting graph is passed back
% for further processing.
p(head3, [t(noun, Noun), c(Ra, La)], G) :- nl, write(head3a),
p(Ra, La, g(Ta, Wa, Laa, Raa, Conf, Gindex)),
root(Noun, Root),
g(T, Root, L, R, Conf2, Gindex2),
attach([([gindex, Gindex, []]), g(T, Root, L, R, Conf2, Gindex2), Gx],
6(T, object),
attach(null, in, Gx, g(Ta, Wa, Laa, Raa, Conf, Gindex), G),
write([head3a]), nl.
%
% head3c => noun head
% ***** This rule is tested by suite sentence #60 *****
% Predicate HEAD3C processes the head structure and retrieves the canonical graph for the root of

Appendix C. Prolog Source Code 119
the noun. If the graph of the noun is a subtype of ACTION or EVENT, then the graph of the noun identifies the function for which the head is an instrument. Therefore, the graph of the noun is attached to the graph of the head structure using the conceptual relation 'purpose.' The resulting graph is passed back for further processing.

p(head3, [[(noun,Noun), c(Rh,Lh), G]], G) :- nl, write(head3c),
p(Rh,Lh,g(Th,Rfh,Lch,Rch,Conf,Gindx)),
root(Noun,Root),
g(Tn,Root,L,R,Conf2,Gindx2),
attach([[gindx,Gindx,[[]],g(Tn,Root,L,R,Conf2,Gindx2),Gx],
(\(\delta\)(Tn,action);\(\delta\)(Tn,event)),
attach([[purp,null,Gx]],G(Th,\(\delta\)Rfh,Lch,Rch,Conf,Gindx),Gt),
write([head3c]), nl.

%
%
head3dh => noun head

***** This rule is tested by suite sentence #15 *****

Predicate HEAD3DH processes the head structure and retrieves the canonical graph for the root form of the noun. This rule checks to see if the graph of the head structure is a subtype of the noun graph. If so, the two words are concatenated together and used as the referent for a concept whose type is the type of the head graph (the more specific type). The resulting graph is passed back for further processing.

p(head3, [[(noun,Noun), c(Rh,Lh), G]], G) :- nl, write(head3dh),
p(Rh,Lh,g(Th,Rfh,Lch,Rch,Conf,Gindx)),
root(Noun,Root),
g(Tn,Root,L,R,Conf2,Gindx2),
attach([[gindx,Gindx,[[]],g(Tn,Root,L,R,Conf2,Gindx2),Gx],
is(a(T,Super),
\(\delta\)(T,Super),
concat_atom([Root,"_",Rfh],Rres),
G = g(Th,Rres,Lch,Rch,Conf,Gindx),
write([head3dh]), nl, !.

%
%
head3dn => noun head

***** This rule is tested by suite sentence #67, #62 *****

Predicate HEAD3DN processes the head structure and retrieves the canonical graph for the root form of the noun. This rule checks to see if the graph of the noun is a subtype of the graph of the head structure. If so, the two words are concatenated together and used as the referent for a concept whose type is the type of the noun graph (the more specific type). The resulting graph is passed back for further processing.

p(head3, [[(noun,Noun), c(Rh,Lh), G]], G) :- nl, write(head3dn),
p(Rh,Lh,g(Th,Rfh,Lch,Rch,Conf,Gindx)),
root(Noun,Root),g(T,Root,L,R,Conf2,Gindx2),
attach([[gindx,Gindx,[[]],g(T,Root,L,R,Conf2,Gindx2),g(Tn,Rnn,Ln,Rn,Conf2,Gindx2)],
is(a(T,Super),
\(\delta\)(T,Super),
concat_atom([Rnn,"_",Rfh],Rres),
G = g(Th,Rres,Ln,Rn,Conf,Gindx),
write([head3dn]), nl, !.

%
%
head3b => noun head
% ***** This rule is tested by suite sentence #1 *****
% Predicate HEAD3B processes the head structure and retrieves the canonical graph for the root of
% the noun. This rule checks to see if the head is an action. If so, then the graph of the noun is
% attached to the graph of the head. The resulting graph is passed back for further processing.
p(head3, [t(noun,Noun), c(Ra,La)], G) :- nl, write(head3b),
    p(Ra,La,g(Ta,Wa,La,Conf,Gindyx)),
    root(Noun,Root),
    g(T,Root,LR,Conf2,Gindyx2),
    attach([g(indx,Gindyx,[]),g(T,Noun,LR,Conf2,Gindyx2),Gx]),
    6(Ta,action),
    attach(_,Gx,g(Ta,Wa,La,Conf,Gindyx),G),
    write([head3b]), nl.
%
%
% head3e => noun head
% ***** This rule is tested by suite sentence #54 *****
% Predicate HEAD3E processes the head structure and retrieves the canonical graph for the root of
% the noun. The graph of the noun to the graph of the head using the conceptual relation 'type.'
% The resulting graph is then passed back for further processing.
p(head3, [t(noun,Noun), c(Rh,Lh)], G) :- nl, write(head3e),
    p(Rh,Lh,g(Th,Rfh,Lch,Rch,Conf,Gindyx)),
    root(Noun,Root),
    g(T,Root,LR,Conf2,Gindyx2),
    attach([g(indx,Gindyx,[]),g(T,Root,LR,Conf2,Gindyx2),Gx]),
    attach([type,null,Gx],g(Th,Rh,Lch,Rch,Conf,Gindyx),G),
    write([head3e]), nl.
%
%
% head4 => id head
% ***** This rule is tested by suite sentence #54 *****
% Predicate HEAD4 processes the head structure and creates a graph for the identifier. If the graph
% of the head is a subtype of object the graph of the identifier is then attached to the graph of the
% head using the conceptual relation 'name.'
p(head4, [t(id,Identifier), c(Ra,La)], G) :- nl, write(head4),
    p(Ra,La,g(Ta,Wa,La,Conf,Gindyx)),
    6(Ta,object),
    attach([name,null,g(id,Identifier,[],[],1,1)],g(Ta,Wa,La,Conf,Gindyx),G),
    write([head4]), nl.
%
%
% ivs1 => to vinf
% ***** This rule is tested by suite sentence #16 *****
% Predicate TVS1 finds the root of the infinitive verb form. The canonical graph of the root is then
% retrieved and the monadic conceptual relation 'to' is attached to it. The resulting graph is passed
% back for further processing.
p(ivs1,[t(to,To),t(vinf,Vinf)],G) :- nl, write(ivs1),
    root(Vinf,Root),
    g(T,Root,LR,Conf,Gindyx),
    attach([g(indx,Gindyx,[]),g(T,Root,LR,Conf,Gindyx),Gx]),
    attach([to,To,Gx,G],write([ivs1],nl.

Appendix C. Prolog Source Code 121
n1 => np
***** This rule is tested by suite sentence #53 *****
Predicate N1 processes the noun phrase and passes the resulting graph back for further
processing.
p(n1,[c(Ra,La)],G) :- nl, write(n1),
p(Ra,La,G),
write([n1]), nl.

n2a => np pp
***** This rule is tested by suite sentence #61 *****
Predicate N2A processes the noun phrase and the prepositional phrase. The graph of the
prepositional phrase is attached to the graph of the noun phrase using the preposition as a literal
role marker. The resulting graph is passed back for further processing.
p(n2,[c(Rnp,Lnp),c(Rpp,Lpp)],G) :- nl, write(n2),
p(Rnp,Lnp,Gnp),
p(Rpp,Lpp,Gpp),
attach([Lpp,_,Gpp,Gnp,G]),
write([n2a]), nl.

n2b => np pp
***** This rule is tested by suite sentence #33 *****
Predicate N2B handles the case where the pp structure uses the pp2 production rule. The rule
processes the noun phrase and the compound prepositional phrase. The graph of each of the
prepositional phrases is attached to the graph of the noun phrase using the prepositions as literal
role markers. The resulting graph is passed back for further processing.
p(n2,[c(Rnp,Lnp),c(Rpp,Lpp)],G) :- nl, write(n2b),
p(Rnp,Lnp,Gnp),
p(Rpp,[Prep1,Prep2],Lpp,Gpp),
attach([Prep1,_,Gpp,Gnp,Gtemp]),
attach([Prep2,_,Gpp,Gtemp,G]),
write([n2b]), nl.

n3 => np rest
***** This rule is tested by suite sentence #58 *****
Predicate N3 processes the np and passes ahead a copy of the resulting graph to be processes
with the relative restrictive clause. A copy of the graph of the noun phrase is also passed back
for further processing. The graph of the restrictive relative clause is not passed back, but added
to the relative graph list during processing of the 'rest' predicates.
p(n3,[c(Rn,Ln),c(Rr,La)],Gn) :- nl, write(n3),
p(Rn,Ln,Gn),
p(Rr,Gn,Lr,Gr),
write([n3]), nl.

n4a => np of n
***** This rule is tested by suite sentence #60 *****
Predicate N4A processes the noun phrase. If the resulting graph is a subtype of ACTION or EVENT then the nominal structure is processed. The graph of the nominal is then attached to the graph of the noun phrase using the structural role marker 'agnt.' The resulting graph is then passed back for further processing.

\[p(n4, c(Rn,Ln), p(Rn,Ln,g(T1,N1,L1,L1,Conf,Conf)), \text{nl}, \text{write}(n4a)), p(Rn,Ln,Gn), attach(obj_{...}, Gn,g(T1,N1,L1,L1,Conf,Conf), G), \text{write}(n4a)), \text{nl}.\]

n4c => np of n

***** This rule is tested by suite sentence #55 *****

Predicate N4C handles the case where the noun phrase is 'a result' in the phrase 'as a result.' The noun phrase to check for 'a result.' If a match is made, the nominal is processed and passed back for further processing.

\[p(n4, c(Rn,Ln), p(Rn,Ln,g(Typ, result, L, R, Conf, Conf)), \text{nl}, \text{write}(n4c)), p(Rn,Ln,Gn), \text{write}(n4c)), \text{nl}.\]

n4b => np of n

***** This rule is tested by suite sentence #61 *****

Predicate N4B processes the noun phrase and the nominal structures. If the noun phrase graph is a subtype of MEASURE, then its graph is attached to the graph of the nominal structure using the conceptual relation 'size.' The resulting graph is passed back for further processing.

\[p(n4, c(Rn,Ln), p(Rn,Ln,g(Tnp, Nnp, Snp, Fnp, Confnp, Gindexp)), \text{nl}, \text{write}(n4b)), p(Rn,Ln,Gn), \text{nl}, \text{write}(n4b)), \text{nl}.\]

n4d => np of n

***** This rule is tested by suite sentence #58 *****

Predicate N4D processes the noun phrase and nominal structures. The graph of the nominal is then attached to the graph of the noun using the literal role marker 'of.' The resulting graph is then passed back for further processing.

\[p(n4, c(Rn,Ln), p(Rn,Ln,Gnp), p(Rn,Ln,Gn), attach(of_{...}, Gnp, G), \text{write}(n4d)), \text{nl}.\]

n5 => np of np pp

***** This rule is tested by suite sentence #7 *****
Predicate N5 processes the first noun phrase and checks to see if its graph is a subtype of ACTION. If so, the second noun phrase and the prepositional phrase are processes. The graph of the prepositional phrase is attached to the graph of the second noun phrase using the leading preposition as a literal role marker. Then, the resulting graph is attached to the graph of the first noun phrase using the structural role marker 'obj.' This graph is then passed back for further processing.

```
p(n5,[c(Ra,La),t(of,of),c(Rb,Lb),c(Rc,Lc)],G) :- nl, write(n5),
p(Ra,La,Ga),
Ga = g(Ta,Na,LLa,RLa,Conf,Gindx),
\(Ta\,(Ta,action),
p(Rb,Lb,Gb),
p(Rc,Pe,Lc,Gc),
attach(Pe,_,Gc,Gb,Gr),
attach(obj,_,Gr,g(action,Na,LLa,RLa,Conf,Gindx),G),
write([n5]), nl.
```

n6 => n6.

***** This rule is tested by suite sentence #62 *****

Predicate N6 processes the nc structure and passes the resulting graph back for further processing.

```
p(n6,[c(Rnc,Lnc)],G) :- nl, write(n6),
p(Rnc,Lnc,G),
write([n6]), nl!.
```

n7 => cl

***** This rule is tested by suite sentence #7 *****

Predicate N7 processes the clause and passes the resulting graph back for further processing.

```
p(n7,[c(Ra,La)],G) :- nl, write(n7),
p(Ra,La,G),
write([n7]), nl.
```

n8 => cl conj cl

***** This rule is tested by suite sentence #17 *****

Predicate N8 processes the two clauses and joins the resulting graphs to a conjunction concept. The conjunction concept is given a type which is the minimal common supertype of the two clause graphs. This conjunction graph is then passed back for further processing.

```
p(n8,[c(Rc1,Lc1),t(conj,Conj),c(Rc2,Lc2)],G) :- nl, write(n8),
p(Rc1,Lc1,G(T1,R1,P1,F1,C1,G1)),
p(Rc2,Lc2,G(T2,R2,P2,F2,C2,G2)),
get_com_sup(T1,T2,Tc),
G=g(Tc,Conj,[1],[Conj,null,g(T1,R1,P1,F1,C1,G1)],(Conj,null,g(T2,R2,P2,F2,C2,G2))],1,1),
write([n8]), nl.
```

n9 => cl, cl conj cl

***** This rule is tested by suite sentence #18 *****

Predicate N9 processes the three clause structures and joins the resulting three graphs to a conjunction concept. The conjunction concept is given a type which is the minimal common
% supertype of the three clause graphs. The resulting conjunction graph is passed back for further
% processing.
p(n9,[c(Rc1,Lc1),t('',''),c(Rc2,Lc2),t(conj,Conj),c(Rc3,Lc3)],G) :- ni, write(n9),
p(Rc1,Lc1,g(T1,R1,P1,F1,C1,G1)),
p(Rc2,Lc2,g(T2,R2,P2,F2,C2,G2)),
p(Rc3,Lc3,g(T3,R3,P3,F3,C3,G3)),
get_com_sup(T1,T2,Ttemp),
get_com_sup(Ttemp,T3,Tc),
G=g(Tc,Conj,[1],[Conj,null,g(T1,R1,P1,F1,C1,G1)],[Conj,null,g(T2,R2,P2,F2,C2,G2)],[Conj,null,g(T3,R3,P3,F3,C3,G3)]],1,1),
write([n9]), nl.

% nc1 => np conj np
% ***** This rule is tested by suite sentence #62 *****
% Predicate NC1 processes the two noun phrase structures and the resulting graphs are joined to a
% conjunction concept. The conjunction concept is given a type which is the minimal common
% supertype of the two noun phrase graphs.
p(nc1,[c(Ra,La),t(conj,Conj),c(Rb,Lb)],G) :- nl, write(nc1),
p(Ra,La,g(Ta,Rla,pla,fla,conf,gindx)),
p(Rb,Lb,g(Tb,Rlb,plb,flb,conf2,gindx2)),
get_com_sup(Ta,Tb,Tc),
G = g(Tc,Conj,[1],[
    (Conj,null,g(Ta,Rla,pla,fla,conf,gindx)),
    (Conj,null,g(Tb,Rlb,plb,flb,conf2,gindx2)])],1,1), !,
write([nc1]), nl.

% nc2 => np , nc
% ***** This rule is tested by suite sentence #20 *****
% Predicate NC2 processes the noun phrase structure followed by the nc structure. The resulting
% graphs are joined to a conjunction concept. The conjunction concept is given a type which is the
% minimal common supertype of the noun phrase graph and compound nominal (nc) graph. The
% conjunction graph is then passed back for further processing.
p(nc2,[c(RnP,LnP),t('',''),c(Rnc,Lnc)],G) :- nl, write(nc2a),
p(RnP,LnP,g(Ta,Ra,Pa,Fa,confa,gindx)),
p(Rnc,Lnc,g(Tb,Rb,Pb,Fb,confb,gindxb)),
get_com_sup(Ta,Tb,Tc),
G = g(Tc,Rb,[1],[
    (Rb,null,g(Ta,Ra,Pa,Fa,confa,gindx)),
    (Rb,null,g(Tb,Rb,Pb,Fb,confb,gindxb)])],1,1), !,
write([nc2a]), nl.

% np1 => predet det adjs head
% ***** This rule is tested by suite sentence #21 *****
% Predicate NP1 processes the head structure and passes its graph forward when the adjective
% sequence is processed. The monadic conceptual relation 'quant' is attached to the resulting graph
% with the referent contained in the variable 'Predet.' The resulting graph is then passed back for
% further processing.
p(np1,[t(predet,Predet),t(det,Det),c(Ra,La),c(Rh,Lh)],G) :- nl, write(np1),
p(Rh,l,h,Gb),
p(Ra,La,Gh,Ga),
attach([[quant,Predet[],],Ga,Gb),
attach([[Det,Det[],],Gb,G),
write([np1]), nl.

% %
% np2 => pred det head
% ***** This rule is tested by suite sentence #22 *****
% Predicate NP2 processes the head structure and the monadic conceptual relation 'quant' is
% attached to the resulting graph with the referent contained in the variable 'Predet.' The resulting
% graph is then passed back for further processing.

p(np2,[t(Predet,Predet),t(Det,Det),c(Rh,Lh),G]) :- nl, write(np2),
p(Rh,Lh,Gh),
attach([[quant,Predet[],],Gh,Gb),
attach([[Det,Det[],],Gb,G),
write([np2]), nl.

% %
% np3 => pred adj head
% ***** This rule is tested by suite sentence #23 *****
% Predicate NP3 processes the head structure and passes its graph forward when the adjective
% sequence is processed. The monadic conceptual relation 'quant' is attached to the resulting graph
% with the referent contained in the variable 'Predet.' The resulting graph is then passed back for
% further processing.

p(np3,[t(Predet,Predet),c(Ra,La),c(Rh,Lh),G]) :- nl, write(np3),
p(Rh,Lh,Gh),
p(Ra,La,Gh,Ga),
attach([[quant,Predet[],],Ga,G),
write([np3]), nl.

% %
% np4 => pred head
% ***** This rule is tested by suite sentence #24 *****
% Predicate NP4 processes the head structure and the monadic conceptual relation 'quant' is
% attached to the resulting graph with the referent contained in the variable 'Predet.' The resulting
% graph is then passed back for further processing.

p(np4,[t(Predet,Predet),c(Rh,Lh),G]) :- nl, write(np4),
p(Rh,Lh,Gh),
attach([[quant,Predet[],],Gh,G),
write([np4]), nl.

% %
% np5 => det ord # adj head
% ***** This rule is tested by suite sentence #25 *****
% Predicate NP5 processes the head structure and passes its graph forward when the adjective
% sequence is processed. The monadic conceptual relation 'attribute' is attached to the resulting
% graph with the referent contained in the variable 'Ord.' Next, the monadic conceptual relation
% 'quantity' is attached with the referent contained in the variable 'Num.' The resulting graph is
% then passed back for further processing.

p(np5,[t(Det,Det),t(Ord,Ord),t(#,Num),c(Ra,La),c(Rh,Lh),G]) :- nl, write(np5),
\[ p(\text{Rh}, \text{Lh}, \text{Gh}), \]
\[ p(\text{Ra}, \text{La}, \text{Gh}, \text{Ga}), \]
\[ \text{attach}([[\text{det}, \text{Det}, []]], \text{Ga}, \text{Gb}), \]
\[ \text{attach}([[\text{attr}, \text{Ord}, []]], \text{Gb}, \text{Gc}), \]
\[ \text{attach}([[\text{quant}, \text{Num}, []]], \text{Gc}, \text{G}), \]
\[ \text{write}(\text{np5}), \text{nl}. \]

% % % np6 => det ord # head
% ***** This rule is tested by suite sentence #26 *****
% Predicate NP6 processes the head structure and the monadic conceptual relation 'attribute' is
% attached to the resulting graph with the referent contained in the variable 'Ord.' Next, the
% monadic conceptual relation 'quantity' is attached with the referent contained in the variable
% 'Num.' The resulting graph is then passed back for further processing.

\[ p(np6, [t(\text{det}, \text{Det}), t(\text{ord}, \text{Ord}), t(\text{#}, \text{Num}), c(\text{Rh}, \text{Lh})], G) :: \text{nl}, \text{write}(\text{np6}), \]
\[ p(\text{Rh}, \text{Lh}, \text{Gh}), \]
\[ \text{attach}([[\text{det}, \text{Det}, []]], \text{Gh}, \text{Gb}), \]
\[ \text{attach}([[\text{attr}, \text{Ord}, []]], \text{Gb}, \text{Gc}), \]
\[ \text{attach}([[\text{quant}, \text{Num}, []]], \text{Gc}, \text{G}), \]
\[ \text{write}(\text{np6}), \text{nl}. \]

% % % np7 => det ord adjs head
% ***** This rule is tested by suite sentence #27 *****
% Predicate NP7 processes the head structure and passes its graph forward when the adjective
% sequence is processed. The monadic conceptual relation 'attribute' is attached to the resulting
% graph with the referent contained in the variable 'Ord.' The resulting graph is then passed back
% for further processing.

\[ p(np7, [t(\text{det}, \text{Det}), t(\text{ord}, \text{Ord}), c(\text{Ra}, \text{La}), c(\text{Rh}, \text{Lh})], G) :: \text{nl}, \text{write}(\text{np7}), \]
\[ p(\text{Rh}, \text{Lh}, \text{Gh}), \]
\[ p(\text{Ra}, \text{La}, \text{Gh}, \text{Ga}), \]
\[ \text{attach}([[\text{det}, \text{Det}, []]], \text{Ga}, \text{Gb}), \]
\[ \text{attach}([[\text{attr}, \text{Ord}, []]], \text{Gb}, \text{G}), \]
\[ \text{write}(\text{np7}), \text{nl}. \]

% % % np8 => det ord head
% ***** This rule is tested by suite sentence #28 *****
% Predicate NP8 processes the head structure and the monadic conceptual relation 'attribute' is
% attached to the resulting graph with the referent contained in the variable 'Ord.' The resulting
% graph is then passed back for further processing.

\[ p(np8, [t(\text{det}, \text{Det}), t(\text{ord}, \text{Ord}), c(\text{Rh}, \text{Lh})], G) :: \text{nl}, \text{write}(\text{np8}), \]
\[ p(\text{Rh}, \text{Lh}, \text{Gh}), \]
\[ \text{attach}([[\text{det}, \text{Det}, []]], \text{Gb}, \text{G}), \]
\[ \text{write}(\text{np8}), \text{nl}. \]

% % % np9 => det # adjs head
% ***** This rule is tested by suite sentence #29 *****
% Predicate NP9 processes the head structure and passes its graph forward when the adjective
% sequence is processed. The monadic conceptual relation 'quantity' is attached with the referent contained in the variable 'Num.' The resulting graph is then passed back for further processing.
p(np9,[t(det,Det),t(#,Num),c(Ra,La),c(Rh,Lh)],G) :- nl, write(np9),  
p(Rh,Lh,Gh),  
p(Ra,La,Gh,Ga),  
attach([(det,Det,[])],Ga,Gb),  
attach([(quant,Num,[])],Gb,G),  
write((np9)), nl.

%
% np10 => det # head  
***** This rule is tested by suite sentence #30 *****  
% Predicate NP10 processes the head structure and the monadic conceptual relation 'quantity' is attached with the referent contained in the variable 'Num.' The resulting graph is then passed back for further processing.
p(np10,[t(det,Det),t(#,Num),c(Rh,Lh)],G) :- nl, write(np10),  
p(Rh,Lh,Gh),  
attach([(det,Det,[])],Gh,Gb),  
attach([(quant,Num,[])],Gb,G),  
write((np10)), nl.

% % np11 => det adj head.  
***** This rule is tested by suite sentence #55 *****  
% Predicate NP11 processes the head structure and passes its graph forward when the adjective sequence is processed. The resulting graph is then passed back for further processing.
p(np11,[t(det,Det),c(Ra,La),c(Rh,Lb)],G) :- nl, write(np11),  
p(Rh,Lb,Gb),  
p(Ra,La,Gb,Ga),  
attach([(det,Det,[])],Ga,G),  
write((np11)), nl.

% % np12 => det head  
***** This rule is tested by suite sentence #53 *****  
% Predicate NP12 processes the head structure and returns the resulting graph for further processing after the determiner contained in the variable 'Det' is attached using the monadic conceptual relation 'det.'
p(np12,[t(det,Det),c(Ra,La)],G) :- nl, write(np12),  
p(Ra,La,Ga),  
attach([(det,Det,[])],Ga,G),  
write((np12)), nl.

% % np17 => # adj head  
***** This rule is tested by suite sentence #31 *****  
% Predicate NP17 processes the head structure and passes its graph forward when the adjective sequence is processed. The monadic conceptual relation 'quantity' is attached with the referent contained in the variable 'Num.' The resulting graph is then passed back for further processing.
p(np17,[t(#,Num),c(Ra,La),c(Rh,Lh)],G) :- nl, write(np17),  
p(Rh,Lh,Gh).

Appendix C. Prolog Source Code 128
p(Ra,La,Gh,Ga),
attach([((quant,Num,[])],Ga,G),
write([np17]), nl.

% %
% np18 => # head
% ***** This rule is tested by suite sentence #62 *****
% Predicate NP18 processes the head structure and the monadic conceptual relation 'quantity' is
% attached with the referent contained in the variable 'Num.' The resulting graph is then passed
% back for further processing.
  p(np18,[t(#,Num),c(Rh,Lb)],G) :- nl, write(np18),
  p(Rh,Lb,Gh),
  attach([((quant,Num,[])],Gh,G),
  write([np18]), nl.
%
%
% np19 => adjs head
% ***** This rule is tested by suite sentence #55 *****
% Predicate NP19 processes the head structure and passes its graph forward when the adjective
% sequence is processed. The resulting graph is then passed back for further processing.
  p(np19,[c(Ra,La),c(Rb,Lb)],G) :- nl, write(np19),
  p(Rb,Lb,Gb),
  p(Ra,La,Gb,G),
  write([np19]), nl.
%
%
% np20 => head
% ***** This rule is tested by suite sentence #54 *****
% Predicate NP20 processes the head structure and passes the resulting graph back for further
% processing.
  p(np20,[c(Ra,La)],G) :- nl, write(np20),
  p(Ra,La,G),
  write([np20]), nl.
%
%
% np22 => pron
% ***** This rule is tested by suite sentence #52 *****
% Predicate NP22 creates a graph for the pronoun and passes it back for further processing.
  p(np22,[t(pron,Pron)],G) :- nl, write(np22),
  G = g(pron,Pron,[]), [],1,1),
  write([np22]), nl.
%
%
% np23 => np ( head )
% ***** This rule is tested by suite sentence #67 *****
% Predicate NP23 hands the cases of a parenthetical head structure. The noun phrase is processed
% first, followed by the head structure. The graph of the head structure is then attached to the graph
% of the noun phrase using the conceptual relation 'name.' The resulting graph is passed back for
% further processing.
  p(np23,[c(Ra,La),t('(',')'),c(Rh,Lh),t(')'),'),G) :- nl, write(np23),
  p(Ra,La,Ga),
p(Rh,Lh,Gh),
attach([[name,null,Gh]],Gz,G),
write([inp23]), nl.

% %
% nvs1 => ven
% ***** This rule is tested by suite sentence #32 *****
% Predicate NVS1 finds the root of the past participle ven. The canonical graph for the root is then
% retrieved and passed back for further processing.
p(nvs1,[t(ven,Ven)],g(T,Ven,CL,RL,Conf,Gindx)) :- nl, write(nvs1),
   root(Ven,Root),
   g(T,Root,CL,RL,Conf,Gindx),
   write([nvs1]), nl.
%
% pp1 => prep n
% ***** This rule is tested by suite sentence #61 *****
% Predicate PP1 processes the nominal structure and passes back the resulting graph for further
% processing. The preposition is also passed back to be used as a literal role marker during later
% attachment.
p(pp1,Prep,[t([prep,Prep],c(Ra,La)],G) :- nl, write(pp1),
   p(Ra,La,G),
   write([pp1]), nl.
%
% pp2 => prep conj prep n
% ***** This rule is tested by suite sentence #33 *****
% Predicate PP2 handles the case where two prepositions precede a nominal structure. The
% nominal structure is processed and the resulting graph is passed back for further processing.
% Since there are two prepositions, a list containing both prepositions is passed back to be used as
% literal role markers during later attachments.
p(pp2,Preps,[t([prep,Prep1],t([conj,Conj]),t([prep,Prep2],c(Rn,Ln)],G) :- nl, write(pp2),
   p(Rn,Ln,G),
   Preps=[Prep1,Prep2],
   write([pp2]), nl.
%
% pred1 => avs
% ***** This rule is tested by suite sentence #57 *****
% Predicate PRED1 is passed the graph Gs which is the subject of the verb. The active verb
% sequence is processed and Gs is attached to the graph of the verb sequence using the structural
% role marker 'agnt.' The resulting graph is passed back for further processing.
p(pred1,Gs,[c(Rv,Lv,Gv)],G) :- nl, write(pred1),
p(Rv,Lv,Gv),
   attach(agnt,Gs,Gv,G),
   write([pred1]), nl.
%
% pred2 => avs n
% ***** This rule is tested by suite sentence #58 *****
% Predicate PRED2 is passed the graph Gs which is the subject of the verb. The active verb
% sequence and the nominal structure are processed. Gs is attached to the graph of the verb
% sequence using the structural role marker 'agent.' The graph of the nominal is attached to the
% graph of the verb sequence using the structural role marker 'obj.' The resulting graph is passed
% back for further processing.
% p(pred2, Gs, [c(Rv,Lv),c(Rn,Ln),G]) :- nl, write(pred2), % avs n
% p(Rv,Lv,Gv),
p(Rn,Ln,Gn),
attach(agt,_,Gs,Gv,Gt),
attach(obj,_,Gn,Gt,G),
write([pred2]), nl.
%
%
%
pred3 => avs n d d

***** This rule is tested by suite sentence #57 *****
% Predicate PRED3 is passed the graph Gs which is the subject of the verb. The active verb
% sequence, the nominal structure, and the adverbial are processed. Gs is attached to the graph of
% the verb sequence using the structural role marker 'agent.' The graph of the nominal is attached to
% the graph of the verb sequence using the structural role marker 'obj.' The graph of the adverbial
% is attached to the graph of the verb sequence using the preposition contained in the variable
% 'Prep' as a literal role marker. The resulting graph is passed back for further processing.
% p(pred3, Gs, [c(Rv,Lv),c(Rn,Ln),c(Rd,Ld),G]) :- nl, write(pred3), % avs n
% p(Rv,Lv,Gv),
p(Rn,Ln,Gn),
p(Rd,Prep,Ld,Gd),
attach(agt,_,Gs,Gv,Ga),
attach(obj,_,Ga,Ga,Gd),
attach(Prep,_, Gd, Go, G),
write([pred3]), nl.
%
%
pred4 => avs n d d

***** This rule is tested by suite sentence #19 *****
% Predicate PRED4 is passed the graph Gs which is the subject of the verb. The active verb
% sequence, the nominal structure, and the two adverbials are processed. Gs is attached to the
% graph of the verb sequence using the structural role marker 'agent.' The graph of the nominal is
% attached to the graph of the verb sequence using the structural role marker 'obj.' The graph of
% the first adverbial is attached to the graph of the verb sequence using the preposition contained
% in the variable 'Prep1' as a literal role marker. Next the graph of the second adverbial is attached
% to the graph of the verb sequence using the preposition contained in the variable 'Prep2' as a
% literal role marker. The resulting graph is passed back for further processing.
% p(pred4, Gs, [c(Rv,Lv),c(Rn,Ln),c(Rd1,Ld1),c(Rd2,Ld2),G]) :- nl, write(pred4),
p(Rv,Lv,Gv),
p(Rn,Ln,Gn),
p(Rd1,Prep1,Ld1,Gd1),
p(Rd2,Prep2,Ld2,Gd2),
attach(agt,_,Gs,Gv,Ga),
attach(obj,_,Gn,Ga,Gb),
attach(Prep1,_, Gd1, Gb, Gc),
attach(Prep2,_, Gd2, Gc, G),
write([pred4]), nl.
% pred5 => avs d
% ***** This rule is tested by suite sentence #3 *****
% Predicate PRED5 is passed the graph Gs which is the subject of the verb. The active verb
% sequence and the adverbial are processed. Gs is attached to the graph of the verb sequence using
% the structural role marker 'agent.' The graph of the adverbial is attached to the graph of the verb
% sequence using the preposition contained in the variable 'Prep' as a literal role marker. The
% resulting graph is passed back for further processing.
p(pred5, Gs, [c(Rv, Lv), c(Rd, Ld)], G) :- nl, write(pred5),
  p(Rv, Lv, Gv),
  p(Rd, Prep, Ld, Gd),
  attach(agent, _, Gs, Gv, Ga),
  attach(Prep, _, Gd, Ga, G),
  write([pred5]), nl.
%
%
% pred6 => avs d d
% ***** This rule is tested by suite sentence #29 *****
% Predicate PRED6 is passed the graph Gs which is the subject of the verb. The active verb
% sequence and the two adverbials are processed. Gs is attached to the graph of the verb sequence
% using the structural role marker 'agent.' The graph of the first adverbial is attached to the graph of
% the verb sequence using the preposition contained in the variable 'Prep1' as a literal role marker.
% Next the graph of the second adverbial is attached to the graph of the verb sequence using the
% preposition contained in the variable 'Prep2' as a literal role marker. The resulting graph is
% passed back for further processing.
p(pred6, Gs, [c(Rv, Lv), c(Rd1, Ld1), c(Rd2, Ld2)], G) :- nl, write(pred6),
  p(Rv, Lv, Gv),
  p(Rd1, Prep1, Ld1, Gd1),
  p(Rd2, Prep2, Ld2, Gd2),
  attach(agent, _, Gs, Gv, Ga),
  attach(Prep1, _, Gd1, G2, Gc),
  attach(Prep2, _, Gd2, Gc, G),
  write([pred6]), nl.
%
%
% pred7 => pv
% ***** This rule is tested by suite sentence #21 *****
% Predicate PRED7 is passed the graph Gs which is the subject of the verb. The passive verb
% sequence is processed and Gs is attached to the graph of the verb sequence using the structural
% role marker 'obj.' The resulting graph is passed back for further processing.
p(pred7, Gs, [c(Rv, Lv)], G) :- nl, write(pred7),
  p(Rv, Lv, Gv),
  attach(obj, _, Gs, Gv, G),
  write([pred7]), nl.
%
%
% pred8 => pv d
% ***** This rule is tested by suite sentence #53 *****
% Predicate PRED8 is passed the graph Gs which is the subject of the verb. The passive verb
% sequence and the adverbial are processed. Gs is attached to the graph of the verb sequence using
% the structural role marker 'obj.' The graph of the adverbial is attached to the graph of the verb
sequence using the preposition contained in the variable 'Prep' as a literal role marker. The resulting graph is passed back for further processing.

\[ p(\text{pred8}, \text{Gn}, [\text{c(Rv,Lv)}, \text{c(Rd,Ld)}], G) :- \text{nl, write(pred8)}, \]
\[ p(Rv,Lv,Gv), \]
\[ p(Rd,\text{Prep,Ld,Gd}), \]
\[ \text{attach(obj,_,Gn,Gv,G)}, \]
\[ \text{attach(Prep, _, Gd, Gt, G)}, \]
\[ \text{write([pred8]), nl.} \]

\[
\begin{align*}
\text{pred9} & \Rightarrow \text{pvs d d} \\
\text{***** This rule is tested by suite sentence #63 *****} \\
\text{Predicate PRED9 is passed the graph Gs which is the subject of the verb. The passive verb sequence and the two adverbials are processed. Gs is attached to the graph of the verb sequence using the structural role marker 'obj.' The graph of the first adverbial is attached to the graph of the verb sequence using the preposition contained in the variable 'Prep1' as a literal role marker.} \\
\text{Next the graph of the second adverbial is attached to the graph of the verb sequence using the preposition contained in the variable 'Prep2' as a literal role marker. The resulting graph is passed back for further processing.} \\
p(\text{pred9}, \text{Gn}, [\text{c(Rv,Lv)}, \text{c(Rd1,Ld1)}, \text{c(Rd2,Ld2)}], G) :- \text{nl, write(pred9)}, \]
\[ p(Rv,Lv,Gv), \]
\[ p(Rd1,\text{Prep1,Ld1,Gd1}), \]
\[ p(Rd2,\text{Prep2,Ld2,Gd2}), \]
\[ \text{attach(obj,_,Gn,Gv,G)}, \]
\[ \text{attach(Prep1, _, Gd1, Gt, G1)}, \]
\[ \text{attach(Prep2, _, Gd2, G1, G)}, \]
\[ \text{write([pred9]), nl.} \]
\end{align*}
\]

\[
\begin{align*}
\text{pred10} & \Rightarrow \text{pvs d d} \\
\text{***** This rule is tested by suite sentence #30 *****} \\
\text{Predicate PRED10 is passed the graph Gs which is the subject of the verb. The passive verb sequence and the three adverbials are processed. Gs is attached to the graph of the verb sequence using the structural role marker 'obj.' The graph of the first adverbial is attached to the graph of the verb sequence using the preposition contained in the variable 'Prep1' as a literal role marker.} \\
\text{Next, the graph of the second adverbial is attached to the graph of the verb sequence using the preposition contained in the variable 'Prep2' as a literal role marker. Finally, the graph of the third adverbial is attached to the graph of the verb sequence using the preposition contained in the variable 'Prep3' as a literal role marker. The resulting graph is passed back for further processing.} \\
p(\text{pred10}, \text{Gn}, [\text{c(Rv,Lv)}, \text{c(Rd1,Ld1)}, \text{c(Rd2,Ld2)}, \text{c(Rd3,Ld3)}], G) :- \text{nl, write(pred10)}, \]
\[ p(Rv,Lv,Gv), \]
\[ p(Rd1,\text{Prep1,Ld1,Gd1}), \]
\[ p(Rd2,\text{Prep2,Ld2,Gd2}), \]
\[ p(Rd3,\text{Prep3,Ld3,Gd3}), \]
\[ \text{attach(obj,_,Gn,Gv,G)}, \]
\[ \text{attach(Prep1, _, Gd1, Gt, G1)}, \]
\[ \text{attach(Prep2, _, Gd2, G1, G2)}, \]
\[ \text{attach(Prep3, _, Gd3, G2, G)}, \]
\[ \text{write([pred10]), nl.} \]
\end{align*}
\]

Appendix C. Prolog Source Code
% pred11 => evs n
% ***** This rule is tested by suite sentence #58 *****
% Predicate PRED11 is passed the graph Gs which is the subject of the verb. The attributive verb
% sequence and nominal structure are processed. Gs is attached to the graph of the verb sequence
% using the structural role marker 'agt.' The graph of the nominal is attached to the graph of the
% verb sequence using the structural role marker 'obj.' The resulting graph is passed back for
% further processing.
p(pred11, Gs, [c(Rv,Lv), c(Rn,Ln)], G) :- nl, write(pred11), % evs n
p(Rv,Lv,Gv),
p(Rn,Ln,Gn),
attach(agt,_,Gs,Gv,Gt),
attach(obj,_,Gn,Gt,G),
write([pred11]), nl.
%
%
pred12 => evs d
% ***** This rule is tested by suite sentence #34 *****
% Predicate PRED12 is passed the graph Gs which is the subject of the verb. The attributive verb
% sequence and adverbial structure are processed. The graph of the adverbial is attached to Gs
% using the preposition contained in the variable 'Prep' as a literal role marker. Gs is then attached
% to the graph of the verb sequence using the structural role marker 'agt.' The resulting graph is
% passed back for further processing.
p(pred12, Gs, [c(Rv,Lv), c(Rd,Ld)], G) :- nl, write(pred12),
p(Rv,Lv,Gv),
p(Rd,Prep,Ld,Gd),
attach(Prep,_,Gd,Gs,Gt),
attach(agt,_,Gt,Gv,G),
write([pred12]), nl.
%
%
pred13 => evs n d
% ***** This rule is tested by suite sentence #35 *****
% Predicate PRED13 is passed the graph Gs which is the subject of the verb. The attributive verb
% sequence, nominal structure, and adverbial structure are processed. Gs is then attached to the
% graph of the verb sequence using the structural role marker 'agt.' The graph of the adverbial is
% attached to the graph of the nominal using the preposition contained in the variable 'Prep' as a
% literal role marker. Next, the graph of the nominal is attached to the graph of the verb sequence
% and the resulting graph is passed back for further processing.
p(pred13, Gs, [c(Rv,Lv), c(Rn,Ln), c(Rd,Ld)], G) :- nl, write(pred13),
p(Rv,Lv,Gv),
p(Rn,Ln,Gn),
p(Rd,Prep,Ld,Gd),
attach(agt,_,Gs,Gv,Gt),
attach(Prep,_,Gd,Gn,Gnt),
attach(obj,_,Gat,Gt,G),
write([pred13]), nl.
%
%
pred14 => evs adjs
% ***** This rule is tested by suite sentence #65 *****
Predicate PRED14 is passed the graph Gs which is the subject of the verb. The attributive verb sequence and adjective sequence are processed. Gs is then passed forward in the processing of the adjective sequence and has the graph of the adjective sequence attached to it. The resulting graph is attached to the graph of the verb sequence using the structural role marker 'agnt.' The resulting graph is then passed back for further processing.

\[ p(\text{pred14, Gs, } [\text{c(Rv,Lv), c(Ra,La)}], G) : nl, \text{write(pred14),} \]
\[ p(Rv,Lv,Gv), \]
\[ p(Ra,La,Gs,Ga), \]
\[ \text{attach(} \text{agnt, Ga,Gv,G)} \]
\[ \text{write([pred14]), nl.} \]

\% ~
\% pred15 \implies evs adjs d
\% **** This rule is tested by suite sentence #65 ****
\% Predicate PRED15 is passed the graph Gs which is the subject of the verb. The attributive verb sequence Gs is then passed forward in the processing of the adjective sequence and has the graph of the adjective sequence attached to it. The resulting graph is attached to the graph of the verb sequence using the structural role marker 'agnt.' This graph then has the graph of the adverbial attached to it using the preposition contained in the variable 'Prep' as a literal role marker. The resulting graph is then passed back for further processing.

\[ p(\text{pred15, Gs, } [\text{c(Rv,Lv), c(Ra,La), c(Rd,Ld)}], G) : nl, \text{write(pred15),} \]
\[ p(Rv,Lv,Gv), \]
\[ p(Ra,La,Gs,Ga), \]
\[ p(Rd,Prep,Ld,Gd), \]
\[ \text{attach(} \text{agnt, Ga,Gv,Gb)} \]
\[ \text{attach(Prep, Gd,Gb,G)} \]
\[ \text{write([pred15]), nl.} \]

\% ~
\% pvs1 \implies be ven
\% **** This rule is tested by suite sentence #61 ****
\% Predicate PVS1 finds the root of the past participle ven. The canonical graph for the root is then retrieved and passed back for further processing.

\[ p(\text{pvs1}, [\text{t(be,Aux), t(ven,Ven)}], Ga) : nl, \text{write(pvs1),} \]
\[ \text{root(Ven, Root),} \]
\[ g(T,Root,CL,RL,Conf,Gindx), \]
\[ \text{attach([g(Gindx,Gindx,[])]), g(T,Root,CL,RL,Conf,Gindx), Ga),} \]
\[ \text{write([pvs1]), nl.} \]

\% ~
\% pvs2 \implies be not ven
\% **** This rule is tested by suite sentence #53 ****
\% Predicate PVS2 finds the root of the past participle ven. The canonical graph for the root is then retrieved. The monadic conceptual relation 'negation' is attached to the canonical graph of the root and the resulting graph is passed back for further processing.

\[ p(\text{pvs2}, [\text{t(be,Aux), t(not,not), t(ven,Ven)}], G) : nl, \text{write(pvs2),} \]
\[ \text{root(Ven, Root),} \]
\[ g(T,Root,CL,RL,Conf,Gindx), \]
\[ \text{attach([g(Gindx,Gindx,[])]), g(T,Root,CL,RL,Conf,Gindx), Gx),} \]
\[ \text{attach([neg,not,[]])}, Gx,G), \]
\[ \text{write([pvs2]), nl.} \]
pvs3 \implies\text{be adv ven}

***** This rule is tested by suite sentence #36 *****

Predicate PVS3 finds the root of the past participle (ven) and retrieves its canonical graph. The canonical graph for the adverb is then retrieved and attached to the graph of the past participle.

The resulting graph is then passed back for further processing.

\begin{verbatim}
p(pvs3,[\{\text{be, Aux} \}, \{\text{adv, Adv} \}, \{\text{ven, Ven} \}], G) :- nl, write(pvs3),
root(Ven, Root),
g(T, Root, CL, RL, Conf, Gindex),
attach([\{\text{gindex, Gindex} \}], g(T, Root, CL, RL, Conf, Gindex, Gx),
g(Ta, Adv, Sa, Fa, Conf, Gindex),
Ga = g(Ta, Adv, Sa, Fa, Conf, Gindexa),
attach(adv, Ga, Gx, G),
write([pvs3]), nl.
\end{verbatim}

pvs4 \implies\text{not adv ven}

***** This rule is tested by suite sentence #37 *****

Predicate PVS4 finds the root of the past participle (ven) and retrieves its canonical graph. The monadic conceptual relation 'negation' is attached to the graph of the past participle. Next, the canonical graph for the adverb is then retrieved and attached to the graph of the past participle.

The resulting graph is then passed back for further processing.

\begin{verbatim}
p(pvs4,[\{\text{be, Aux} \}, \{\text{not, not} \}, \{\text{adv, Adv} \}, \{\text{ven, Ven} \}], G) :- nl, write(pvs4),
root(Ven, Root),
g(T, Root, CL, RL, Conf, Gindex),
attach([\{\text{gindex, Gindex} \}], g(T, Root, CL, RL, Conf, Gindex, Gx),
attach([\{\text{neg, not} \}], Gx, Gy),
g(Ta, Adv, Sa, Fa, Conf, Gindexa),
Ga = g(Ta, Adv, Sa, Fa, Conf, Gindexa),
attach(adv, Ga, Gv, G),
write([pvs4]), nl.
\end{verbatim}

pvs5 \implies\text{mod bei ven}

***** This rule is tested by suite sentence #55 *****

Predicate PVS5 finds the root of the past participle ven. The canonical graph for the root is then retrieved. The monadic conceptual relation 'modal' is attached to the canonical graph of the root and the resulting graph is passed back for further processing.

\begin{verbatim}
p(pvs5,[\{\text{mod, Mod} \}, \{\text{bei, Aux} \}, \{\text{ven, Ven} \}], G) :- nl, write(pvs5),
root(Ven, Root),
g(T, Root, CL, RL, Conf, Gindex),
attach([\{\text{gindex, Gindex} \}], g(T, Root, CL, RL, Conf, Gindex, Gx),
attach([\{\text{mod, Mod} \}], Gx, G),
write([pvs5]), nl.
\end{verbatim}

pvs6 \implies\text{mod adv bei ven}

***** This rule is tested by suite sentence #54 *****

Predicate PVS6 finds the root of the past participle (ven) and retrieves its canonical graph. Next, the canonical graph for the adverb is then retrieved and attached to the graph of the past...
participle. Finally, the monadic conceptual relation 'modal' is attached to the graph of the past participle. The resulting graph is then passed back for further processing.

p(pvs6,[t(mod,Mod),t(adv,Adv),t(bei,Aux),t(ven,Ven)],G) :- nl, write(pvs6),
root(Ven,Root),
g(T,Root,CL,RL,Conf,Gindx),
attach([(gindx,Gindx,[])],g(T,Root,CL,RL,Conf,Gindx),Gx),
g(Ta,Adv,Sa,Fa,Conf,Gindx),
Ga=g(Ta,Adv,Sa,Fa,Conf,Gindx),
attach(adv_,Ga,Gx,Gv),
attach([(mod,Mod,[[]])],Gv,G),
write((pvs6)), nl.

pvs7 => mod not bei ven

***** This rule is tested by suite sentence #16 *****

Predicate PVS7 finds the root of the past participle ven. The canonical graph for the root is then retrieved. The monadic conceptual relation 'negation' is then attached to the canonical graph of the root. Next, the monadic conceptual relation 'modal' is attached to the canonical graph of the root and the resulting graph is passed back for further processing.

p(pvs7,[t(mod,Mod),t(not,Not),t(bei,Aux),t(ven,Ven)],G) :- nl, write(pvs7),
root(Ven,Root),
g(T,Root,CL,RL,Conf,Gindx),
attach([(gindx,Gindx,[])],g(T,Root,CL,RL,Conf,Gindx),Gx),
attach([(neg,Not,[[]])],Gx,Ga),
attach([(mod,Mod,[[]])],Ga,G),
write((pvs7)), nl.

pvs8 => mod bei adv ven

***** This rule is tested by suite sentence #15 *****

Predicate PVS8 finds the root of the past participle (ven) and retrieves its canonical graph. Next, the canonical graph for the adverb is then retrieved and attached to the graph of the past participle. Finally, the monadic conceptual relation 'modal' is attached to the graph of the past participle. The resulting graph is then passed back for further processing.

p(pvs8,[t(mod,Mod),t(bei,Aux),t(adv,Adv),t(ven,Ven)],G) :- nl, write(pvs8),
root(Ven,Root),
g(T,Root,CL,RL,Conf,Gindx),
attach([(gindx,Gindx,[])],g(T,Root,CL,RL,Conf,Gindx),Gx),
g(Ta,Adv,Sa,Fa,Conf,Gindx),
Ga=g(Ta,Adv,Sa,Fa,Conf,Gindx),
attach(adv_,Ga,Gx,Gb),
attach([(mod,Mod,[[]])],Gb,G),
write((pvs8)), nl.

pvs9 => mod not bei adv ven

***** This rule is tested by suite sentence #38 *****

Predicate PVS9 finds the root of the past participle (ven) and retrieves its canonical graph. The monadic conceptual relation 'negation' is then attached to the graph of the past participle. Next, the canonical graph for the adverb is then retrieved and attached to the graph of the past participle. Finally, the monadic conceptual relation 'modal' is attached to the graph of the past participle.
The resulting graph is then passed back for further processing.

p(pvs9,[t(mod,Mod),t(not,Not),t(bei,Aux),t(adv,Adv),t(ven,Ven)],G) :- nl, write(pvs9),
root(Ven,Root),
g(T,Root,CL,RL,Conf,Gindx),
attach([g(gindx,Gindx,[[]]),g(T,Root,CL,RL,Conf,Gindx),Gx),
attach([neg,Not,[[]],Gx,Gv),
g(Ta,Adv,Sa,Confa,Gindx),
Ga=g(Ta,Adv,Sa,Confa,Gindx),
attach(adv,_,Ga,Gv,Gb),
attach([mod,Mod,[[]]],Gb,G),
write([pvs9]), nl.

rest1 => rcon pred

***** This rule is tested by suite sentence #58 *****

Predicate REST1 passes the graph Gn forward for the processing of the predicate structure. The
resulting graph is added to the relative graph list. The predicate check_rel checks to see if the
graph being added to the relative graph list has already been added. If it is not the last graph in
the list, then it is appended to the end of the relative graph list. This function is necessary to
prevent multiple copies of the same graph appearing on the relative graph list due to
backtracking done by Prolog. A copy of the graph added to the relative graph list is also passed
back for future processing.

p(rest1,Gn,[t(rcon,Rcon),c(Rp,lp)],Gp) :- nl, write(rest1),
p(Rp,Gn,lp,Gp),
retract(relative(L)),
check_rel(L,Gp,K),
asserta(relative(K)),
write([rest1]),nl.

rest2 => cl

***** This rule is tested by suite sentence #58 *****

Predicate REST2 processes the clause structure and attaches the graph Gn to the resulting graph
using the structural role marker 'obj.' The resulting graph is added to the relative graph list. The
predicate check_rel checks to see if the graph being added to the relative graph list has already
been added. If it is not the last graph in the list, then it is appended to the end of the relative
graph list. This function is necessary to prevent multiple copies of the same graph appearing on
the relative graph list due to backtracking done by Prolog. A copy of the graph added to the
relative graph list is also passed back for future processing.

p(rest2,Gn,[c(Rc,lc)],G) :- nl, write(rest2),
p(Rc,lc,Gp),
attach(obj,_,Gn,Gp,G),
retract(relative(L)),
check_rel(L,G,K),
asserta(relative(K)),
write([rest2]),nl.

s1 => ss

***** This rule is tested by suite sentence #53 *****

Predicate S1 processes the ss structure and the resulting graph is a complete conceptual graph for
% the sentence
p(s1,[c(Ra,La),t(.,.)],G) :- nl, write(s1),
p(Ra,La,G),
write([s1]), nl.
%
% s2 => sp
% ***** This rule is tested by suite sentence #60 *****
% Predicate S2 processes the sp structure and the resulting graph is a complete conceptual graph
% for the sentence.
p(s2,[c(Rsp,sp),t(.,.)],G) :- nl, write(s2),
p(Rsp,sp,G),
write([s2]), nl.
%
% s3 => sc
% ***** This rule is tested by suite sentence #55 *****
% Predicate S3 processes the sc structure and the resulting graph is a complete conceptual graph
% for the sentence.
p(s3,[c(Rsc,sc),t(.,.)],G) :- nl, write(s3),
p(Rsc,sc,G),
write([s3]), nl.
%
% sc1 => ss conj ss
% ***** This rule is tested by suite sentence #55 *****
% Predicate SC1 processes the two simple sentences (ss) and attaches the resulting graphs to a
% conjunction concept. The conjunction concept is given a type which is the minimal common
% supertype of the two simple sentence graphs. The resulting conjunction graph is passed back for
% further processing.
p(sc1,[c(Rss1,Lss1),t(conj,Conj),c(Rss2,Lss2)],G) :- nl, write(sc1),
p(Rss1,Lss1,Gs1),g(Ts1,----)=Gs1,
p(Rss2,Lss2,Gs2),g(Ts2,----)=Gs2,
get_com_sup(Ts1,Ts2,Conj),
G=g(Conj,Conj,1),[Conj,null,Gs1],(Conj,null,Gs2)],1,1),
write((sc1)), nl.
%
% sp1 => d , n pred conj pred
% ***** This rule is tested by suite sentence #39 *****
% Predicate SP1 processes the adverbial structure, nominal structure, and the first predicate
% structure. The a copy of the adverbial graph is attached to the graph of the first predicate
% structure using the contents of the variable 'Prep' as a literal role marker. The second predicate
% structure is then processed. Another copy of the adverbial graph is attached to the graph of the
% second predicate structure using the contents of the variable 'Prep' as a literal role marker. The
% two predicate graphs are then attached to a conjunction concept which is given a type that is the
% minimal common supertype of the two predicate graphs. The resulting conjunction graph is
% passed back for further processing.
p(sp1,[c(Rd,La),t(,,),c(Rn,Ln),c(Rpa,Lpa),t(conj,Conj),c(Rpb,Lpb)],G) :- nl, write(sp1),
p(Rd,Prep,Ld,Gd),
p(Rn,Ln,Gn),


p(Rpa,Gn,Lpa,g(Ta,Rla,Pla,Fla,Conf1,Gindx1)),
attach(Prep,_,Gd,g(Ta,Rla,Pla,Fla,Conf1,Gindx1),Gv1),
p(Rpb,Gn,Lpb,g(Tb,Rlb,Plb,Flb,Conf2,Gindx2)),
attach(Prep,_,Gd,g(Tb,Rlb,Plb,Flb,Conf2,Gindx2),Gv2),
get_com_sup(Ta,Tb,Tc),
G = g(Tc,Conj,[(Conj,null,Gv1),(Conj,null,Gv2)],1,1),
write((sp1)),nl.

%
%
% sp2 => n pred conj pred
%
% ***** This rule is tested by suite sentence #60 *****
%
% Predicate SP2 processes the nominal and forwards its conceptual graph for the processing of the
% two predicate structures. The graphs of the predicate structures are attached to a conjunction
% concept. The conjunction concept is given a type which is the minimal common supertype of
% the two simple sentence graphs. The resulting conjunction graph is passed back for further
% processing.
p(sp2,[c(Rn,Ln),c(Rpa,Lpa),G(Conj,Conj),c(Rpb,Lpb)],G) :- nl, write(sp2),
p(Rn,Ln,Gn),
p(Rpa,Gn,Lpa,g(Ta,Rla,Pla,Fla,Conf,Gindx)),
p(Rpb,Gn,Lpb,g(Tb,Rlb,Plb,Flb,Conf2,Gindx2)),
get_com_sup(Ta,Tb,Tc),
G = g(Tc,Conj,[(Conj,null,g(Ta,Rla,Pla,Fla,Conf,Gindx)),
               (Conj,null,g(Tb,Rlb,Plb,Flb,Conf2,Gindx2))],1,1),
write(sp2)),nl.
%
%
% ss1 => d, n pred
%
% ***** This rule is tested by suite sentence #40 *****
%
% Predicate SS1 processes the adverbial structure followed by the nominal structure. The graph of
% the nominal is then passed forward for the processing of the predicate structure. The graph of
% the adverbial structure is then attached to the graph of the predicate structure using the contents
% of the variable 'Prep' as a literal role marker. The resulting graph is passed back for further
% processing.
p(ss1,[c(Rd,Ld),G(Prep,_,Gd,_,Gs)],G) :- nl, write(ss1),
p(Rd,Prep,Ld,Gd),
p(Ra,La,Ga),
p(Rb,La,Gb,Gs),
attach(Prep,_,Gd,Gs,G),
write((ss1)),nl.
%
%
% ss2 => n pred
%
% ***** This rule is tested by suite sentence #53 *****
%
% Predicate SS2 processes the nominal structure and then passes its conceptual graph forward for
% the processing of the predicate structure. The resulting predicate graph is then passed back for
% further processing.
p(ss2,[c(Ra,La),c(Rb,Lib)],G) :- nl, write(ss2),
p(Ra,La,Ga),
p(Rb,La,Gb,G),
write((ss2)),nl.

Appendix C. Prolog Source Code
% xvs1 => to bei ven
% ****** This rule is tested by suite sentence #58 ******
% Predicate XVS1 finds the root of the past participle ven and then retrieves its canonical graph
% from the canon. This graph is then passed back for further processing.
p(xvs1,[t(To,To),t(bei,Be),t(ven,Ven)],Gx):- nl, write(xvs1),
    root(Ven,Root),
    g(T,Root,CL,RL,Conf,Gindx),
    attacha([gindx,Gindx,[]],g(T,Root,CL,RL,Conf,Gindx),Gx),
    write([xvs1]), nl.


%**************************************************
% %
% MAIN.PL
% %
% This file, MAIN.PL, contains the graph processing routines as well
% as the attach and select predicates. This is the core of the semantic
% analyzer, with predicates to consult all the other required files.
% %
% This program analyzes parse trees of sentences and produces conceptual graphs from them. The
% program employs a knowledge base consisting of a type hierarchy (isa) and a list of canonical
% graphs g(R,T,P,F,Conf,Gindx) where T is the head concept of the graph, R is the referent of this
% head concept, P is the list of slots of the head concept and F is the list of relations and filler
% concepts successfully attached to the head concept. Conf is the variable that has been installed
% to implement a confidence level associated with the conceptual graphs. Gindx is the variable
% used for the canonical graph index, which is used by the LINKER and GENERATOR of the
% ASPIN system.
% %
% To use this program use the goal proc(N), where N is an integer identifying a parse tree.
% %
%**************************************************
% %
% ****** SUGGESTED CHANGES: ******
% Standardize format for p(). Number and order of arguments.
% Distinguish forced and selective attach predicates by name.
% Return result and relative graphs through global lists: results and leftovers.
% Add predicate to generate standard CG's from g().
%   ************************************************************
% %
% Process a parse tree node c(R,L) using predicate p(R,L,F), where
% R is the id of the rule that produced the constituent,
% L is the list of daughter constituent nodes, and
% G is the semantic structure (conceptual graph) that is associated with node c(R,L).
% p(R,[c(Ra,La),...c(Rn,Ln)],G) :-
%   p(Ra,La,Ga),...,p(Rn,Ln,Gn), % Process daughter nodes
%   comb(Ga,...,Gn,G), % Combine daughter graphs
% Note that in some cases, combining is interleaved with the processing.
In predicates p(R, Gs, L, G), Gs is the graph of a leading constituent (sentence subject) to be
pushed to a lower level for attachment. For example in sentence rule s2, the subject nominal, n,
is passed down to the clause that processes the predicate. The predicate processing clause
attaches the nominal to the graph returned from the verb sequence clause.

The graph data structure used in this program can only handle trees, so relative clauses that
attach to the main clause by a role are treated by duplicating the common concept and placing
the relative clause graph in a list. Graphs in this list can be joined later by an exact match join.

To use this program, enter proc(#), which will look for an s1-type constituent whose first
argument is # (parse tree number) and try to construct a graph, Graph, from it.

The structure used here as a shorthand (canonical) conceptual graph is
g(W, T, L, R, Conf, Gindx) where
    W is the referent of the head of the structure
    T is the type of the head of the structure
    L is the list of possible cases or slots (Rel, MarkL, TypL)
        Rel is the type of conceptual relation
        MarkL is the list of acceptable markers (prepositions) for that slot
        TypL is the list of acceptable concept types for the filler of the slot
    R is the list of filled cases or slots
    Conf is the confidence level number
    Gindx is the canonical graph index number
Note that 'null' denotes the absence of a role marker.

Consult statements to compile the other semantic analyzer files
:- consult('roots.pl').
:- consult('canon.pl').
:- consult('semrules.pl').
:- consult('suite.pl').

Define empty list of relative (graphs of relative clauses).
:- dynamic relative/1.
relative([]).

Define SELECT(M, R, T, A, L, C) predicate which looks for an element of the possible slots (L) of
the graph to be attached to, such that the marker (M) is a member of the list of acceptable
markers for the slot and R is the relation type, concept type (T) of the graph (A) to fill the slot is
a member of the list of acceptable concept types in L. Then, a conceptual relation triple, (Rel,
M, A) is formed and sent back as (C) for attachment. If an acceptable slot is not found in L,
then if (M) is a condition marker, a (cond) slot is formed and sent for attachment a result marker,
a (result) slot is formed and sent back

This select predicate succeeds if the attaching graph can fill the first available slot of graph to
which it is being attached.
select(M, Rel, T, A, [(Rel, ML, TL) | Z], C) :-
    member(M, ML),
member(X, TL),
\delta(T, X),
C = (Rel, M, A).
%
%
% This select predicate tries inference if the role marker of the attaching graph is valid, but the
% types don't match. The inference tried by this rule is based on the type of the attaching graph.
select(M, Rel, T, A, [(Rel, ML, TL) \| Z], C) :-
member(M, ML),
know(X, Krel, T),
member(X, TL),
C = (Rel, M, g(X, '!', [], [(Krel, null, A)], 1, 1)).
%
%
% This select predicate tries inference if the role marker of the attaching graph is valid, but the
% types don't match. The inference tried by this rule is based on the actual concept of the
% attaching graph.
select(M, Rel, T, g(T, Con_name, _, _, _, _), [(Rel, ML, TL) \| Z], C) :-
member(M, ML),
know(X, Krel, Con_name),
member(X, TL),
C = (Rel, M, g(X, '!', [], [(Krel, null, g(T, Con_name, _, _, _, _))], 1, 1)).
%
%
% The select predicate tries the next empty slot of the graph to which the attachment is trying to be
% made.
select(M, R, T, A, [Y|Z], C) :-
%
%
% This select predicate checks a literal role marker to see if the graph can be attached using the
% conceptual relation 'condition' since there are no more slots in L.
select(M, _, T, A, [], C) :-
member(M, [if, when]),
member(X, [time, state, action, event, be, '!']),
\delta(T, X),
C = (cond, M, A).
%
%
% This select predicate checks for the literal role marker 'as a result'. If it is present, the graph is
% attached using the conceptual relation 'condition'.
select('as a result', _, T, g(T, Wa, CLa, FLa, Conf, Gindex), [], C) :-
C = (cond, 'as a result of', g(T, Wa, CLa, FLa, Conf, Gindex)).
%
%
% This select predicate checks for the literal role marker 'before'. If present, the graph is attached
% using the conceptual relation 'before'.
select(M, _, T, A, [], C) :-
member(M, [before]),
(\delta(T, time) \| \delta(T, state) \| \delta(T, action) \| \delta(T, event) \| \delta(T, act_struct) ),
!, C = (before, M, A).

Appendix C. Prolog Source Code
This predicate gets the supertype of the graph being attached and goes through the select process again using the graph's supertype. Note that the only effect this will have is to check a different set of inference facts.

\[ \text{select}(M, \text{Rel}, T, A, [(\text{Rel}, M, L, TL) \mid Z], C) :~ \text{isa}(T, Tsup), \]
\[ \text{select}(M, \text{Rel}, Tsup, A, [(\text{Rel}, M, L, TL) \mid Z], C). \]

Predicate attach\((M, \text{Role}, Ga, Gh, Gr)\) forms a slot with \(M\) and \(Ga\) based on the list of potential slots of \(Gh\), and then attaches this slot to the list of filled slots of \(Gh\).

This special attach rule is used for the case of simple adverb attachment. In order for this rule to be attempted, the marker passed to the attach predicate must be 'adv'

\[ \text{attach}(\text{adv}, \text{Role}, g(Ta, \text{Adv}, Sa, Fa, Confa, Ginxa), g(Th, Wh, CLh, FLh, Conf2, Gindx2), g(Th, Wh, CLh, FLr, Conf2, Gindx2)) :~ \text{nl}, \text{write}(\text{adv}_{-}\_\text{attach}), \]
\[ \text{select}(\text{adv}, \text{Role}, Th, g(Ta, \text{Adv}, [], Fa, Confa, Ginxa), Sa, Slot), !, \]
\[ \text{append}(\text{FLh}, [\text{Slot}], \text{FLr}), \]
\[ \text{write}([\text{adv}_{-}\_\text{attach}]), \text{nl}. \]

This is the standard attach\(5\) rule which calls the set of select rules to determine an appropriate slot or relation by which to attach the two graphs.

\[ \text{attach}(M, \text{Role}, g(Ta, Wa, CLa, FLa, Conf, Gindx), g(Th, Wh, CLh, FLh, Conf2, Gindx2), g(Th, Wh, CLh, FLr, Conf2, Gindx2)) :~ \text{select}(M, \text{Role}, Ta, g(Ta, Wa, CLa, FLa, Conf, Gindx), CLh, Slot), \]
\[ \text{append}(\text{FLh}, [\text{Slot}], \text{FLr}). \]

This is the force attach\(3\) that does not check the select rules, but always succeeds in attaching a graph \(F\) to the graph \(g(T, C, L, R, \text{Conf}, \text{Gindx})\). Predicate attach\([(\text{Rel}, M, G)]\) attaches the list of slots \([(\text{Rel}, M, G)]\) to the input graph Gin to form Gout.

\[ \text{attach}(F, g(T, C, L, R, \text{Conf}, \text{Gindx}), g(T, C, L, S, \text{Conf}, \text{Gindx})) :~ \text{append}(R, F, S). \]

The predicate `go` will process a file that contains all the parse trees for a given sentence. All the trees will be analyzed, and the tree with the highest confidence will be written to a file. Note that although this routine exists in the analyzer, the methodology to determine the best graph has not yet been devised.

\[ :- \text{dynamic treecounter}/1. \]
\[ \text{treecounter}(1). \]
\[ \text{go:- retract(treecounter}(N)), \]
\[ \text{mproc}(N), \]
\[ \text{NS is N+1,} \]
\[ \text{assert(treecounter}(NS)), \]
\[ \text{go}. \]
\[ \text{go:- findbest,} \]
\[ \text{retract(best}(N, Bconf)), \text{nl}, \text{nl}, \]

Appendix C. Prolog Source Code 144
printstring("The best graph is . . . "), nl,
retract(graph(N,G)), cger(G), assert(treecounter(1)).
%
\texttt{mproc(N) :- s(N,c(ST,L)), init, p(ST,L,G),}
\texttt{nl, write(done), nl,nl,printstring(""""), wp(L),printstring(""""),nl,}
\texttt{nl, write(mainResultGraph), nl, cger(G),}
\texttt{nl, write(relativeGraphs), nl, relative(RL), shof(RL),}
\texttt{assert(graph(N,G)), assert(relgraph(N,RL)), !.}
\texttt{mproc(N) :- s(N,c(ST,L)), assert(graph(N,g(null,null,[],[],0,0)),}
\texttt{assert(relgraph(N,[])).
%}
\texttt{:- dynamic best/2.}
\texttt{:- dynamic gcounter/1.}
\texttt{best(0,0).}
\texttt{gcounter(0).}
\texttt{findbest :- best(N,Bconf),}
\texttt{retract(gcounter(C)),}
\texttt{X is C+1,}
\texttt{graph(X,g(Typ,Rel,Cl,Rl,Conf,Gindx)),}
\texttt{checkbest(N,Bconf,X,Conf,Newbest,Newconf),}
\texttt{retract(best(N,Bconf)),}
\texttt{assert(best(Newbest,Newconf)),}
\texttt{assert(gcounter(X)),}
\texttt{findbest.}
\texttt{findbest.}
%
\texttt{checkbest(N,Bconf,X,Conf,Newbest,Newconf) :-}
\texttt{Conf > Bconf, Newbest=X,}
\texttt{Newconf=Conf!.}
\texttt{checkbest(N,Bconf,X,Conf,Newbest,Newconf) :- Newbest=N, Newconf=Bconf.
%***************************************************************************
%***************************************************************************
% This predicate invokes the semantic analysis of parse tree # N.
%***************************************************************************
%***************************************************************************
%
\texttt{proc(N) :- s(N,c(ST,L)),}
\texttt{init,}
\texttt{p(ST,L,G),}
\texttt{nl, write(done), nl,nl,}
\texttt{printstring(""""), wp(L),printstring(""""),nl,nl,}
\texttt{write(mainResultGraph), nl, cger(G),}
\texttt{nl, write(relativeGraphs), nl, relative(RL), shof(RL).}
%
%***************************************************************************
% This predicate initializes the relative graph and leftover lists
init :-
\texttt{retract(relative(R)),}
\texttt{assert(relative([])).
%
% This predicate causes converts each graph in its argument list to a standard form conceptual
% graph.
shof([]).
shof([H:T]) :- cger(H),nl, shof(T).
%
%
% This program takes a graph g/4 and transforms it into a standard conceptual graph format. The
% concepts are not listed in numerical order, but this can be corrected by a reordering routine.
cger(G) :- asserta(cno(1)), cout((1,G)).
%
gout([]).
gout([H:T]) :- cout(H),gout(T).
%
cout((Cnum, g(Typ,Ref,CL,Roles,Conf,Gindx))) :- % Print the head concept of G.
    nl, prntstrng( "[ ",
    write(Cnum), prntstrng(" : "),
    write(Typ), prntstrng(" : "),
    write(Ref), prntstrng(" ]"),
    rout(Roles, [], Z),
    reverse(Z, Y),
    gout(Y).
%
%
% List the relations.
rout([],Z,Y).
rout([Rtyp,Rmark,[]], Z, GLin, GLout) :-
    nl, prntstrng(" -> ",
    write(Rtyp), prntstrng(" : "),
    write(Rmark), prntstrng(" )" ),
    rout(Z, GLin, GLout).
rout([Rtyp,Rmark,G], Z, GLin, GLout) :-
    cno(Cnum),
    Nnum is Cnum + 1,
    retract(cno(Cnum)),
    asserta(cno(Nnum)),
    nl, prntstrng(" -> ",
    write(Rtyp), prntstrng(" : " ),
    write(Rmark), prntstrng(" ) -> [ " ),
    write(Nnum), prntstrng(" ]" ),
    append([Nnum,G], GLin, GLtemp),
    rout(Z, GLtemp, GLout).
%
%
% The following line 'load_files' consults a library file 'strings.pl' that is necessary for the
% concatenation that is done the semantic rules head3dh and head3dn.
%
:- load_files(library(strings)).
%
%
% The predicates wp, and vp print the sentence out after analysis.
%
wp([]).
wp([X1,X2,X3]) :-
vp(X), vp(Z).
vp(X):- t(A,B)=X, write(B), printstring(" ").
vp(X):- c(R,L)=X, wp(L).
%
%
% Predicate `check_rel(L,Gp,K)` checks the relative list L to see if Gp has already been added -
% this is possible due to the recursive nature of prolog. Gp will already be attached if a semantic
% rule with multiple definitions fails. When this happens, the relative graph list needs to be
% checked to make sure multiple copies of Gp are not added. If Gp is the last element of the
% relative graph list, Gp is not added. If Gp is not the last element of L, then it is appended to L.
% The cut (!) is needed to prevent backtracking that will append Gp to L.
%
check_rel(L,Gp,K):-
    last(Gp,L), !, K=L, write(islast).
check_rel(L,Gp,K):-
    append(L, [Gp], K), write(notlast).
%
%
% Predicate `last(X,Y)` succeeds if X is the last element of the list Y. This predicate is used by
% check_rel/3 above.
%
last(X,[X]).
last(X,[_|Y]) :- last(X,Y).
%
%
% Predicate member/3 as defined in Clocksin and Meliises book "Programming in Prolog"
%
member(X,[X|_]).
member(X,[_|Y]) :- member(X,Y).
%
%
*****PORTABILITY NOTE*****
% The function append/3 is need not be explicitly expressed in Quiatus Prolog for the Sun
% Sparcstaion, but it must on Quintus Prolog on the Apollo Workstaion
%
% append([],L,L).
% append([X|L1],L2,[X|L3]):- append(L1,L2,L3).
%
%
printstring([]).
printstring([H|T]):- put(H), printstring(T).
%
%
reverse([],[]). reverse([X|Y],Z):-
    reverse(Y,W),
    append(W,[X],Z).
%
%
% %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

Appendix C. Prolog Source Code 147
This file, CANON.PL, contains the conceptual type hierarchy information as well as the canonical graphs. It also contains definitions for the routines to find the minimal common supertype and check whether a graph is a subtype of a given type.

Below, comparable type relation predicates are declared.

\[ \text{op}(500, \text{XfY}, \text{isa}). \]
\[ \text{op}(500, \text{XfY}, \text{isa}). \]

Two types are compared in the type hierarchy using the predicate \( \text{isa} \) or \( \text{isa} \) \( Y \)

Note that \( \text{isa} \) is the universal type.

\( \text{isa} \) object.

\( \text{isa} \) universal.

The unknown id type refers to an object.

The unknown pronoun type refers to a subtype of the universal type.

\( \text{isa}(\text{id}, \text{X}) \) \( \text{isa} \) \( \text{X} \) \( \text{object} \).

\( \text{isa}(\text{pron}, \text{X}) \) \( \text{isa} \) \( \text{X} \) \( \text{universal} \).

\( \text{isa}(\text{X}, \text{Y}) \) checks to see if \( \text{X} \) is a subtype of \( \text{Y} \). Below the \( \text{isa} \) predicate is defined.

\( \text{isa}(\text{X}, \text{X}) \) \( \text{isa} \) \( \text{X} \).

\( \text{isa}(\text{X}, \text{Y}) \) \( \text{isa} \) \( \text{Y} \).

\( \text{isa}(\text{X}, \text{Z}) \) \( \text{isa} \) \( \text{Y} \) \( \text{Z} \).

Below the predicate \( \text{get_com_sup/3} \) is defined. When given two types, \( \text{Ta} \) and \( \text{Tb} \), it determines the minimal common supertype.

\( \text{get_com_sup}(\text{Ta}, \text{Tb}, \text{Com_sup}) \) \( \text{isa} \) \( \text{Ta} \) \( \text{Com_sup} \) \( \text{Tb} \).

\( \text{get_com_sup}(\text{Ta}, \text{Tb}, \text{Com_sup}) \) \( \text{isa} \) \( \text{Ta} \) \( \text{Com_sup} \) \( \text{Tb} \).

\( \text{get_com_sup}(\text{Ta}, \text{Tb}, \text{Com_sup}) \) \( \text{isa} \) \( \text{Tb} \) \( \text{Com_sup} \) \( \text{Ta} \).

\( \text{get_com_sup}(\text{Ta}, \text{Tb}, \text{Com_sup}) \) \( \text{isa} \) \( \text{Ta} \) \( \text{Supa} \) \( \text{get_com_sup}(\text{Supa}, \text{Tb}, \text{Com_sup}) \).

CONCEPT TYPE HIERARCHY

(General concept definitions)

Of the form \( \text{X isa Y} \) which states that \( \text{X} \) is a subtype of \( \text{Y} \)

processor \( \text{isa} \) device.
logic \( \text{isa} \) device.
memory \( \text{isa} \) device.
logic_memory \( \text{isa} \) device.
carrier \( \text{isa} \) device.
transducer \( \text{isa} \) device.
command \( \text{isa} \) value.
address \( \text{isa} \) value.
data \( \text{isa} \) value.
constant \( \text{isa} \) value.
information isa value.
device isa object.
value isa object.
object isa universal.
be isa state.
read isa move.
write isa move.
connect isa move.
operate isa action.
move isa action.
action isa universal.
state isa universal.
characteristic isa universal.
physic_struct isa structure.
data_struct isa structure.
act_struct isa structare.
structure isa universal.
affect isa event.
call isa event.
cause isa event.
terminate isa event.
event isa universal.
mem_measure isa measure.
time isa measure.
measure isa universal.
name isa universal.
human isa universal.
%
%*****************************************************************************
%  CONCEPT TYPE HIERARCHY
% (specific concept definitions)
%*****************************************************************************
%
0 isa constant.
1 isa constant.
accept isa read.
access isa read.
accomplish isa action.
accumulate isa action.
accumulator isa memory.
acknowledge isa write.
act isa action.
activate isa action.
add isa operate.
addend isa data.
adder isa logic.
address isa action.
affect isa event.
allow isa action.
also isa characteristic.
alter isa affect.
alu isa logic.
and isa operate.
apply isa write.
are isa be.
argument isa data.
array isa data struct.
arrive isa action.
assembler isa command.
assert isa write.
attempt isa action.
augend isa data.
average isa operate.
begin isa cause.
bit isa mem measure.
branch isa call.
breakpoint isa address.
buffer isa memory.
bush isa carrier.
byte isa mem measure.
cable isa carrier.
cache_1 isa memory.
cache_2 isa action.
change isa affect.
chip isa device.
circuit isa device.
circuitry isa device.
clear isa write.
clock isa device.
code isa command.
code isa information.
coincide isa state.
compare isa operate.
complete isa terminate.
component isa device.
compute isa operate.
computer isa processor.
contain isa physic struct.
content isa information.
continue isa state.
control isa affect.
controller isa processor.
convert isa operate.
isac(co-processor,processor).
count isa data.
counter isa logic memory.
cpu isa processor.
cycle isa act struct.
decode isa operate.
decoder isa logic.
decrement isa operate.
delete isa write.
destroy isa affect.
detect isa action.
detector isa transducer.
determine isa action.
difference isa data.
direct isa affect.
disable isa affect.
disk isa memory.
display isa transducer.
divide isa operate.
dividend isa data.
divisor isa data.
drive isa write.
eprom isa memory.
element isa device.
emit isa action.
enable isa affect.
enter isa call.
eprom isa memory.
equal isa state.
execute isa operate.
fetch isa read.
fill isa action.
flag isa information.
'flip-flop' isa memory.
follow isa act_struct.
force isa action.
function isa action.
gate isa logic.
generate isa action.
halt isa terminate.
handle isa action.
have isa physic_struct.
hold isa physic_struct.
i80486 isa processor.
improve isa affect.
increment isa operate.
index isa address.
indicate isa state.
indicator isa transducer.
initalize isa write.
inivate isa cause.
input_1 isa read.
input_2 isa carrier.
input_3 isa information.
instruction isa command.
interpret isa action.
interrupt isa call.
is isa state.
issue isa action.
jump isa call.
keyboard isa transducer.
label isa address.
latch_1 isa read.
latch_2 isa memory.
level isa information.
line isa carrier.
link isa carrier.
load isa write.
locate isa action.
location isa address.
m8611 isa processor.
machine isa processor.
macro isa command.
make isa cause.
manipulate isa action.
message isa information.
mhz isa measure.
microcomputer isa processor.
microcontroller isa processor.
microprocessor isa processor.
isa(mode,state).
modify isa affect.
multiplexer isa logic.
multiplicand isa data.
multiplier isa data.
next isa act_struct.
occur isa action.
occur isa event.
'op-code' isa command.
operand isa data.
operate isa action.
order isa structure.
output_1 isa information.
output_2 isa write.
output_3 isa carrier.
overflow isa state.
parameter isa data.
pass isa write.
perform isa operate.
peripheral isa device.
period isa time.
pin isa carrier.
point isa action.
pointer isa address.
poll isa read.
port isa memory.
procedure isa command.
product isa data.
produce isa action.
program isa command.
prom isa memory.
pulse_1 isa information.
pulse_2 isa action.
push isa write.
quotient isa data.
ram isa memory.
receive isa operate.
record isa write.
register isa memory.
release isa operate.
remain isa state.
remainder isa data.
remove isa operate.
represent isa action.
request isa call.
require isa action.
reset isa write.
result isa data.
resume isa cause.
return isa call.
rise isa event.
rom isa memory.
rotate isa operate.
routine isa command.
run isa action.
save isa write.
selector isa logic.
send isa move.
sensor isa transducer.
service isa action.
set isa write.
shift isa operate.
signal isa information.
software isa command.
specify isa action.
stack_1 isa data_struct.
stack_2 isa operate.
startup isa cause.
step isa action.
stop isa terminate.
storage isa memory.
store isa write.
strobe_1 isa information.
strobe_2 isa action.
subprogram isa command.
subroutine isa command.
subtracting isa data.
sum isa data.
switch isa logic.
system isa device.
take isa read.
tcycle isa act_struct.
terminal isa transducer.
test isa operate.
timer isa logic_memory.
total isa measure.
transfer isa move.
translate isa operate.
transmit isa write.
unit isa device.
use isa action.
variable isa data.
vector isa mem_measure.
wire isa carrier.

% %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% % THE CANONICAL GRAPHS %
% %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% of the form g(T,N,L,F,Conf,Gindx)
% where:
% T is the type of the graph
% N is the name of the concept
% L is the list of empty slots
% F is the list of filled slots
% Conf is the confidence number
% Gindx is the graph index
% %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% g(0,0,[],[],1,1).
g(1,1,[],[],1,1).
g(mem_measure,16-bit,[state],[adj],[data]),(type,[adj],[device]),[],[],1,1).
g(mem_measure,16-bit,[size],[adj],[data]),[],[],1,2).
g(mem_measure,3-bit,[size],[adj],[data]),(type,[adj],[device]),[],[],1,1).
g(mem_measure,3-bit,[size],[adj],[state]),[],[],1,2).
g(physical_struct,3-input,[attr],[adj],[device]),[],[],1,1).
g(mem_measure,8-bit,[size],[adj],[data]),(type,[adj],[device]),[],[],1,1).
g(mem_measure,8-bit,[size],[adj],[state]),[],[],1,2).
g(mem_measure,'8k',[size],[adj],[processor]),[],[],1,1).
g(mem_measure,'8k',[size],[adj],[memory]),[],[],1,2).
g(accept,accept,[agent],[agent],[by],[device]),(obj,[obj],[data]),[],[],1,1).

Appendix C. Prolog Source Code
g(cache_2,cache,[[obj,[agt,agnt],[value]],[loc,[in],[memory]]],[],1,1).
g(call,call,[[agt,[agt,by],[command]]],[obj,[obj,to],[command]]),[],1,1).
g(call,call,[[agt,[agt,by],[command]]],[obj,[obj,to],[address]]),[],1,2).
g(cause,cause,[[agt,[agt,by],[action]],[obj,[obj],[action]]],[],1,1).
g(cause,cause,[[agt,[agt,by],[value]],[obj,[obj],[action]]],[],1,2).
g(change,change,[[agt,[agt,by],[devicel],[obj],[obj],[value]]],[],1,1).
g(change,change,[[obj,[obj],[agt],[value]],[val,[from],[value]],[rslt,[to],[value]]],[],1,2).
g(chip,chip,[[[],1,1,1],
g(circuit,circuit,[[],1,1,1],
g(circuitry,circuitry,[[],1,1,1],
g(clear,clear,[[agt,[agt,by],[command]]],[obj,[obj],[agt],[value]]],[],1,1).
g(clear,clear,[[agt,[agt,by],[command]]],[obj,[obj],[agt],[value]]],[],1,2).
g(clock,clock,[[in,[of],[in],[device]]],[loc,[on],[device]]],[],1,1).
g(code,code,[[],1,1,1],
g(information,information,[[],1,1,2],
g(coincide,coincide,[[obj,[agt],[action]],[acc,[obj],[with],[value]]],[],1,1).
g(coincide,coincide,[[obj,[agt],[action]],[acc,[obj],[with],[value]]],[],1,2).
g(command,command,[[],1,1,1],
g(compare,compare,[[agt,[agt,by],[device]]],[obj,[obj],[value]],[opnd,[to],[with],[value]]],[],1,1).
g(complete,complete,[[agt,[agt,by],[device]]],[obj,[obj],[agt],[action]]],[],1,1).
g(complete,complete,[[agt,[agt,by],[command]]],[obj,[obj],[agt],[action]]],[],1,2).
g(component,component,[[],1,1,1],
g(compute,compute,[[agt,[agt,by],[logic]]],[obj,[obj],[value]]],[],1,1).
g(compute,compute,[[agt,[agt,by],[action]]],[obj,[obj],[value]]],[],1,2).
g(computer,computer,[[type,[adj],[measure]]],[],1,1).
g(act_struct,act_struct,[[type,[adj],[command]]],[],1,1).
g(act_struct,act_struct,[[type,[adj],[action]]],[],1,2).
g(connect,connect,[[obj,[obj],[device]]],[agt,[agt,by],[action]],[inst,[using],[by],[carrier]],[dest,[to],[with],[device]]],[],1,1).
g(connect,connect,[[agt,[agt,by],[carrier]]],[obj,[obj],[device]],[dest,[to],[with],[device]]],[],1,2).
g(contain,contain,[[agt,[agt],[memory]]],[obj,[obj],[value]]],[],1,1).
g(contain,contain,[[agt,[agt],[value]]],[obj,[obj],[value]]],[],1,2).
g(contain,contain,[[agt,[agt,by],[device]]],[obj,[obj],[device]]],[],1,3).
g(content,content,[[in,[of],[in],[memory]]],[],1,1).
g(continue,continue,[[agt,[agt,by],[device]]],[obj,[obj],[agt],[action]],[loc,[at],[information]]],[],1,1).
g(control,control,[[agt,[agt,by],[device]]],[obj],[device]],[obj],[],1,1).
g(control,control,[[agt,[agt,by],[device]]],[obj],[device]],[obj],[device]],[[],1,1,2).
g(control,control,[[agt,[agt,by],[device]]],[obj],[device]],[obj],[device]],[[],1,1,2).
g(co-processor,co-processor,[[],1,1,1],
g(characteristic,correctly,[[man,[adv],[action]]],[],1,1).
g(characteristic,correctly,[[man,[adv],[event]]],[],1,2).
g(count,count,[[of],[of],[operate]]],[],1,1).
g(counter,counter,[[in],[of],[in],[device]]],[],1,1).
g(cpu,cpu,[[in],[of],[in],[device]]],[],1,1).
g(cycle,cycle,[[],1,1,1],
g(data,data,[[loc,[in],[on],[memory]]],[src,[from],[device]],[dest,[to],[device]]],[],1,1).
g(data,data,[[loc,[in],[on],[carrier]]],[src,[from],[device]],[dest,[to],[device]]],[],1,2).
g(data,data,[[loc,[in],[on],[logic_memory]]],[src,[from],[device]],[dest,[to],[device]]],[],1,3).
g(decode,decode,[[agt,[agt,by],[device]]],[obj,[obj],[value]]],[],1,1).
g(decoder, decoder, [type, [adj], [measure]], [], 1, 1).
g(decrement, decrement, [[agt, [grnt, by], [device]], [obj, [agtnt, by], [value]]], [], 1, 1).
g(decrement, decrement, [[agt, [grnt, by], [event]], [obj, [agtnt, by], [value]]], [], 1, 1).
g(delete, delete, [[agt, [grnt, by], [command]], [obj, [obj], [value]]], [], 1, 1).
g(delete, delete, [[agt, [grnt, by], [write]], [obj, [obj], [value]]], [], 1, 1, 1).
g(destroy, destroy, [[agt, [grnt, by], [device]], [obj, [obj], [value]]], [], 1, 1, 1).
g(destroy, destroy, [[agt, [grnt, by], [action]], [obj, [obj], [value]]], [], 1, 1, 1, 1).
g(detect, detect, [[agt, [grnt, by], [device]], [obj, [obj], [value]]], [], 1, 1, 1).
g(detect, detect, [[agt, [grnt, by], [command]], [obj, [obj], [value]]], [], 1, 1, 1, 1).
g(detect, detect, [[agt, [grnt, by], [command]], [obj, [obj], [state]]], [], 1, 1, 1, 3).
g(determine, determine, [[agt, [grnt, by], [device]], [obj, [obj], [value]]], [], 1, 1, 1).
g(determine, determine, [[agt, [grnt, by], [value]], [obj, [obj], [if], [that], [whether], [action]]], [], 1, 1, 2).
g/device, [device], [], 1, 1, 1).
g(difference, difference, [[of, [grnt, [work]]], [of, [carry]]], [], 1, 1).
g(direct, direct, [[agt, [grnt, by], [device]], [obj, [obj], [action]]], [dest, [to], [memory]], [], 1, 1, 1).
g(direct, direct, [[agt, [grnt, by], [command]], [obj, [obj], [action]]], [dest, [to], [memory]], [], 1, 1, 1, 2).
g(disable, disable, [[obj, [obj], [action]]], [agt, [grnt, by], [action]], [], 1, 1, 1).
g(disable, disable, [[obj, [obj], [action]]], [agt, [grnt, by], [command]], [], 1, 1, 1, 2).
g/disk, [disk], [], 1, 1, 1).
g/display, display, [], [], 1, 1, 1).
g/divide, divide, [[agt, [grnt, by], [value]], [obj, [obj], [value]]], [], 1, 1, 1).
g/dividend, dividend, [[of, [of], [operate]]], [], 1, 1, 1).
g/divisor, divisor, [[of, [of], [operate]]], [], 1, 1, 1).
g/drive, drive, [[agt, [grnt, by], [device]], [obj, [obj], [carrier]]], [], 1, 1, 1).
g/characteristic, easily, [[man, [adv], [action]]], [], 1, 1, 1).
g/characteristic, easily, [[man, [adv], [event]]], [], 1, 1, 1, 2).
g/eeprom, eeprom, [], [], 1, 1, 1).
g/element, element, [], [], 1, 1).
g/emit, emit, [[agt, [grnt, by], [carrier]], [obj, [obj], [value]]], [], 1, 1, 1).
g/enable, enable, [[obj, [obj], [action]]], [agt, [grnt, by], [action]], [], 1, 1, 1, 1).
g/enable, enable, [[agt, [grnt, by], [value]], [obj, [obj], [device]]], [], 1, 1, 1, 1, 2).
g/enter, enter, [[agt, [grnt, by], [processor]], [obj, [obj], [state]]], [], 1, 1, 1, 1).
g/eeprom, eeprom, [], [], 1, 1, 1).
g/equal, equal, [[agt, [grnt, by], [value]], [obj, [obj], [value]]], [], 1, 1, 1).
g/event, event, [], [], 1, 1, 1).
g/execute, execute, [[agt, [grnt, by], [processor]], [obj, [obj], [value]]], [], 1, 1, 1, 1).
g/fetch, fetch, [[agt, [grnt, by], [processor]], [obj, [obj], [value]]], [], 1, 1, 1).
g/fetch, fetch, [[agt, [grnt, by], [command]], [obj, [obj], [value]]], [], 1, 1, 1, 2).
g/fill, fill, [[agt, [grnt, by], [value]], [obj, [obj], [value]]], [], 1, 1, 1, 1, 1).
g/fill, fill, [[agt, [grnt, by], [device]], [obj, [obj], [value]]], [src, [from], [memory]], [], 1, 1, 1, 2).
g/fill, fill, [[agt, [grnt, by], [command]], [obj, [obj], [value]]], [src, [from], [memory]], [], 1, 1, 1, 3).
g/flag, flag, [], [], 1, 1, 1).
g/flop-flop, flop-flop, [], [], 1, 1, 1).
g/follow, follow, [[agt, [grnt, by], [action]], [obj, [obj], [value]]], [], 1, 1, 1, 1).
g/force, force, [[agt, [grnt, by], [device]], [obj, [obj], [value]]], [dest, [on], [of], [carrier]]], [], 1, 1, 1, 1).
g/force, force, [[agt, [grnt, by], [device]], [obj, [obj], [device]]], [], 1, 1, 1, 2, 2).
g/force, force, [[agt, [grnt, by], [command]], [obj, [obj], [device]]], [], 1, 1, 1, 3, 3).
g/function, function, [[agt, [grnt, by], [of], [action]], [obj, [obj], [value]]], [], 1, 1, 1, 1).
g/gate, gate, [[type, [adv], [measure]]], [], 1, 1, 1, 1).
g/generate, generate, [[agt, [grnt, by], [device]], [obj, [obj], [value]]], [], 1, 1, 1, 1).
% KNOWLEDGE FOR INFERENCEING
% defined know(X,REL,Y) where Y fills the role of REL for X.
know(value,loc,memory).
know(action,agt,counter).
know(data,size,mem_measure).
know(value,size,mem_measure).
know(value,loc,logic_memory).

Appendix C. Prolog Source Code
Appendix D. Suite of Test Sentences

This appendix contains the suite of test sentences that exercise all of the semantic rules.

1. A memory write stores the data.
2. The counter can increment or decrement.
3. The computer can also operate in a high-speed mode.
4. The system will not operate in 16-bit mode.
5. The system may not correctly operate at 15 mhz.
6. The chip has an on-chip, 8k cache.
7. Resetting stops the execution of the instruction in memory.
8. Resetting before the system stops deletes the startup program.
9. Caching the disk in memory improves performance.
10. The interrupts to be handled are serviced in order.
11. The data to be stored in memory is incremented.
12. The processor is not a 16-bit device.
13. The memory chip cannot be a 16-bit device.
14. The gate may also be a 3-input device.
15. The system memory can be easily accessed.
16. The instruction to execute cannot be fetched.
17. Resetting and initializing clear the memory.
18. Resetting, initializing and loading clear the memory.
19. The machine can perform a read or write asynchronously if the rdwrite signal is high.
20. The i80486 contains a cache, processor and co-processor.
21. All the internal registers are cleared.
22. All the registers are cleared when the system resets.
23. All internal registers are cleared when the system resets.
24. All registers are cleared when the system resets.
25. The first 2 low input signals cause initialization.
26. The first 8 bytes of data are stored in memory.
27. The first low input signal causes initialization.
28. The first byte of data is stored.
29. The 2 low input signals initialize asynchronously if strb is low.
30. The 8 bytes are stored asynchronously if strb is low when the data arrives.
31. 1 priority interrupt is serviced before the next cycle.
32. Reset is complete before the cycle begins.
33. Data to and from the memory is cached.
34. The data is on the bus.
35. The result is a high signal on the output.
36. The counter is asynchronously reset.
37. The buffer is not asynchronously cleared.
38. The counter can not be asynchronously reset.
39. Before the next instruction is executed, the program counter is read and incremented.
40. Before the data is read, the internal register is cleared.
41. The system reset is completed before the cycle begins.
42. The data to be incremented by the counter is read.
43. The counter to increment the data is loaded when LD is high.
44. The data to increment is read from the bus.
45. The counter is to increment the data.
46. The counter to increment the data before conversion is loaded when LD is high.
47. The data to be incremented by the counter is read from the bus.
48. The last step is to increment the data in the register before a reset.
49. The last step is to increment before a reset.
50. The counter cleared asynchronously resets.
51. The counter reset before the cycle begins is cleared.
52. It is cleared asynchronously.
53. The remaining cpu registers are not pushed onto the stack.
54. The -int pin can also be polled with branch instructions.
55. These bits can be tested by a program and specific action can be taken as a result of their state.
56. The -int pin may also be tested using the conditional jump instruction.
57. The hsi records times when events occur.
58. The program counter is a 16-bit register that contains the address of the next instruction to be executed.
59. Stacking the cpu registers, setting the low bit and vector fetching requires a total of 11 cycle periods for completion.
60. Execution of a startcnt instruction connects the ti input pin to the counter input and enables the counter.
61. The output of the clock on ti is disabled by reset of the processor.
62. The 8048 has 27 lines which can be used for input functions or output functions.
63. The 8-bit data is loaded into the -acc register when -strb rises.
64. A rise of the -inc signal increments the data in the -acc register.
65. Buffen is high when ds1 is high and nds2 is low.
66. The data in -acc is applied to -do when -buifen is high.
67. The m6811 is an 8-bit, hcmos microcontroller unit (mcu).
68. The loaded internal registers are pushed onto the stack.
69. The internal 8-bit registers are pushed onto the stack.
70. All loaded, 8-bit registers are pushed onto the stack.
Appendix E. Catalog of Conceptual Relations

All the conceptual relations that are used in this work are listed below. The conceptual graphs are read "the relation of concept 1 is concept 2." For example, the graph below is read "the location of the value is memory."

\[ \text{[ VALUE : value ]} \rightarrow (\text{location}) \rightarrow \text{[ MEMORY : memory ]} \]

The catalog is of the following form.

relation: (abbreviation): \textit{definition}

The conceptual relation is shown in bold, its abbreviation is shown in parentheses, and its definition is shown in italics. An example of the usage is then given below the definition.

accompaniment: (acc): \textit{a device or value that accompanies the object}

[DATA] \leftarrow (\text{obj}) \leftarrow [ARRIVE] \rightarrow (\text{acc : with}) \rightarrow [\text{PARITY}]

after: (after): \textit{the action or event completes after the interval or event}

[START] \rightarrow (after) \rightarrow [\text{INITIALIZE}]

agent: (agtnt): \textit{the device that performs the action}

[INCREMENT] \rightarrow (agtnt) \rightarrow [\text{COUNTER}]
and: (and): conjunction relation indicating subparts
[READ] <- (and) <- [MOVE : and] -> (and) -> [WRITE]

attribute: (attr): a characteristic of the object
[GATE] -> (attr) -> [HCMOS]

before: (before): the action or event completes before the interval or event
[INITIALIZE] ->(before) -> [START]

condition: (cond): the condition that triggers the action or event
[CLEAR] -> (cond : if) -> [RISE] -> (obj) -> [SIGNAL]

destination: (dest): the destination object of an action
[WRITE] -> (dest) -> [MEMORY]

determiner: (det): a word that points to a noun
[REGISTER] -> (det : the)

frequency: (freq): the frequency with which an action recurs
[OPERATE] -> (freq) -> [MHZ]

in: (in): indicates the state of containment of an object by a compatible object
[VALUE] -> (in) -> [MEMORY]

instrument: (inst): the device to implement the action or event
[CONNECT] -> (inst) -> [BUS]

location: (loc): location of a stored value
[VALUE] -> (loc) -> [MEMORY]
**modal**: (mod): an auxiliary that modifies a verb

[READ] -> (mod : can)

**name**: (name): the name of an object

[SIGNAL] -> (name) -> [ID : STRB]

**negation**: (neg): indicates action does not occur or condition does not exist

[RESET] -> (neg)

**object**: (obj): the concept that is the object of an action or event

[READ] -> (obj) -> [VALUE]

**operand**: (opnd): a value used by the operation

[ADD] -> (opnd) -> [VALUE]

**purpose**: (purp): the purpose of a device or action

[USE] -> (purp : for) -> [RESET]

**quantity**: (quant): the number of objects

[REGISTERS] -> (quant : 8)

**result**: (rslt): condition or value resulting from an action or event

[CONVERT] -> (rslt) -> [VALUE]

**size**: (size): the size of an object

[DATA] -> (size) -> [MEASURE : 8-bit]

**source**: (src): the source of an object of an action or event

[READ] -> (obj) -> [VALUE] -> (src) -> [MEMORY]
**type:** (type): *a description of the type of object*

[GATE] -> (type) -> [NAND]

**value:** (val): *the value held by a device or value*

[SIGNAL] -> (val) -> [CONSTANT: high]
Vita

Rob Greenwood was born in Wilmington, Delaware on May 30, 1966. After completing his high school education at Newark High School in June of 1984, he began a 5 year engineering program at David Lipscomb College in Nashville, TN. After completing the requirements at David Lipscomb College, he transferred to Tennessee Technological University in September of 1987 to complete his degree in Electrical Engineering. In August of 1988, he received a Bachelor of Science degree from David Lipscomb College and the following June received his Bachelor of Science in Electrical Engineering from Tennessee Tech.

After completing a B.S. degree in Electrical Engineering, Rob decided to continue his education at Virginia Polytechnic Institute and State University and pursue a M.S. degree in Electrical Engineering. At Virginia Tech he concentrated his studies in the digital design and VLSI design area of Computer Engineering.

After completing a M.S. degree in Electrical Engineering at Virginia Polytechnic Institute and State University, Rob began employment at Motorola Inc., in Austin Texas in October of 1992.

Rob Greenwood