THE COMPUTERIZED COMPREHENSIBILITY SYSTEM MAINTAINER'S GUIDE

David E. Kieras

University of Michigan

Technical Communication Program
Technical Information Design and Analysis Laboratory
2360 Bonisteel Blvd.
Ann Arbor, MI 48109-2108

Technical Report No. 33 (TR-90/ONR33)

September 12, 1990

This research was supported by the Office of Naval Research, Cognitive Science Program, under Contract Number N00014-90-J-4110, Contract Authority Identification Number NR 442f002. Reproduction in whole or part is permited for any purpose of the United States Government.

Abstract

The Computerized Comprehensibility System (CCS) generates a critique of the comprehensibility of a technical document, using techniques and results from AI and cognitive psychology. This report is a guide for the maintainer or developer of CCS. It is not intended as a tutorial on the mechanisms used in CCS, but to allow the qualified programmer to rapidly understand the internal mechanisms of CCS in order to correct, modify, or extend the grammar and criticism rules in CCS. This report contains the following sections: An overview of CCS, a description of each of the components of the CCS system, including the syntax for the various representations used in the system, a series of maintenance examples in which the parser and semantics module of the system are extended to handle new kinds of structures and new criticism rules are added to CCS, and a complete list and explanation of the criticisms produced by the current version.

1. INTRODUCTION

The Computerized Comprehensibility System (CCS) has been described in earlier reports (Kieras, 1985a, 1987, 1989) in terms of the concepts and principles involved. This document is intended to be a guide for the maintainer or developer of CCS. It is assumed that the user of this document is a skilled artificial intelligence programmer, and has knowledge of, or access to, the standard literature in artificial intelligence describing natural language processing and production systems. This document is not intended as a tutorial on the mechanisms used in CCS; its intent is to allow the qualified programmer to rapidly understand the internal mechanisms of CCS in order to correct, modify, or extend the grammar and criticism rules that it uses.

1.1 General Properties of the LISP Code

CCS is written in COMMON LISP, relying on facilities defined in Steele (1984). The LISP code in CCS is generally very heavily commented. For purposes of this document, it should suffice just to point out the top-level information, such as the “philosophy” of the individual modules, and special things to look out for in maintaining or modifying the code. Thus this document will not present much information about the details of the algorithms involved.

The code underwent extensive debugging, and many useful tracing facilities have been left in place. These are usually invoked by global variables that when set to non-NIL will cause the functions to print out useful traces. The examples provided later in this document will give some examples of these traces.

Some of the major modules, in particular the parser and the production system interpreter, have been defined using the package facility to keep their internal symbols separate from those in the rest of the system. However most of the system operates in the USER package. In particular, the lexicon, grammar network, and production rules use symbols defined in the USER package.

1.2 Overview of CCS

The overall structure of CCS and the corresponding files are shown in Figure 1. An input document, stored in a text file, is first broken into individual sentences by a *preprocessor*, and each sentence is then analyzed by a syntactic parser, the Augmented Transition Network Interpreter, *ATN interpreter*, which outputs a parse tree of the sentence. This parse tree is used by the *semantics module* to construct a semantics structure that follows a set of conventions based on Anderson's (1976) ACT semantic network structure. This structure represents the basic propositional content of the sentence. It is processed by a *reference resolution module* which examines the content of the sentence to see whether the objects referred to by the sentence have already been defined previously in the text, and updates the representations of the sentence propositions accordingly. The resulting information is then added to sentence memory (SM). A set of production rules, the *criticism rules*, examines the contents of SM and the contents of passage memory (PM), which contains the content of previously analyzed sentences. The semantics module, in addition to generating the propositional representation, also includes information about various syntactic features of the original sentence, such as whether it was written in the passive voice. The criticism rules can thus examine the semantic content of the sentence in conjunction with its syntactic form, and in relation to the previous contents of the passage. Thus for example, it can detect an *appropriate* use of the passive voice, in which the surface subject of the sentence is the same as the current topic of discussion. The criticism rules generate comments which are written into an output file along with a copy of the input sentences. After the criticism rules have been applied, another set of production rules, the *integration rules,* moves the content of sentence memory into passage memory, and sentence memory is then emptied. The preprocessor then finds the next sentence in the input text and the process continues. At the end of the input, the system prints out other information that it has accumulated along the way, such as the structure of topics and subtopics.

Figure 1 also shows some additional components of CCS that are used to develop the system. The parser interpreter uses an Augmented Transition Network (ATN) which is generated by the *HLG compiler* from a *High*

Level Grammar specification language (HLG). This makes it possible to specify and extend a large and complex grammar relatively easily. The production rules are interpreted by a production rule interpreter, which operates on a representation constructed by a production rule compiler. The interpreter and compiler are both components of a subsystem known as the *Parsimonious Production System* (PPS).

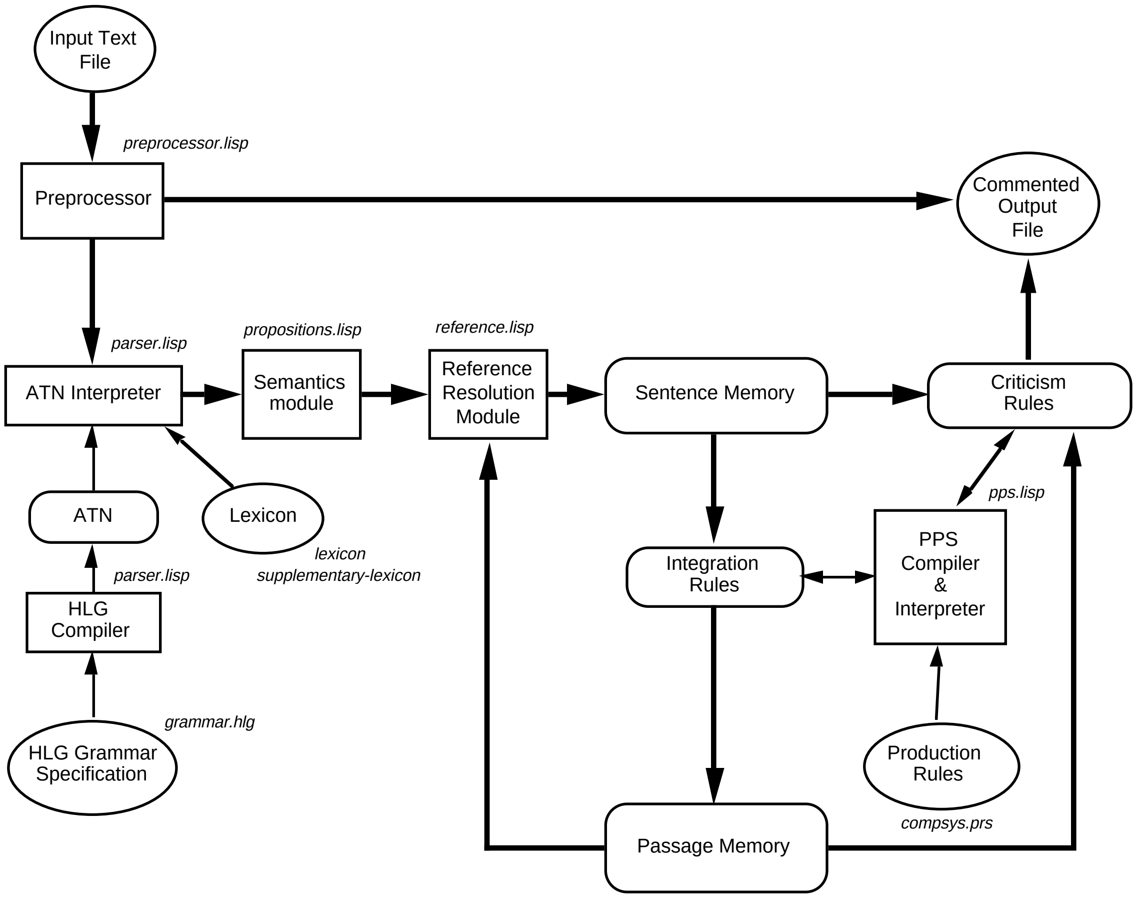

Figure 1. Overall structure of the Computerized Comprehensibility System, CCS. Rectangles are major program modules; ovals are input files. Rounded rectangles are representations constructed by the program. The flow of information from the input text is shown with thick arrows. Thin arrows show other control and information flow. Filenames for the program modules and input files are shown in italics.

1.3 Organization of this Document

Section 2 of this document describes each of the components of the CCS system. The syntax for the HLG grammar specification, PPS production rules, and lexicon entries will be described, along with information about the various representations that CCS constructs. Section 3 consists of a series of maintenance examples in which the parser and semantics module of the system are extended to handle new kinds of structures, and new criticism rules added to CCS. Section 4 is a descriptive list of the criticisms and comments currently generated by CCS.

2. SYSTEM DESCRIPTION

2.1 The Top Level

The top level functions in CCS are contained in the file "ccs-funcs.lisp". There are two basic functions; PROCESS-FILE analyzes a text file, and PROCESS-SENTENCES prompts the user for individual sentences expressed in LISP form, analyzes the sentence, and displays the results. This function is intended for development and debugging, and the file-input function is for the normal mode of use. Each of these functions is called by another set of small top-level functions which allow one to exercise different parts of the system for development and debugging. These functions are described in the maintenance examples below, but the normal usage function is intended to be CRITICIZE-FILE.

Each of these top-level functions can take an optional list of output control options. These are symbols defined in the "ccs-funcs.lisp" file, where they are heavily commented. The intention is to provide different levels of verbosity and detail in the output. The normal output for a user is written into a file with the extension ".out", while more detailed information goes into a file with the extension ".detail"; finally a dribble-like facility is also available as well. The options control how much information is written into these files; for example one can request that no grammatical analyses be written into the ".out" file, while full parse trees for failed parses can be written into the ".detail" file. There is also a provision for displaying and outputting lists of words whose grammatical categories were guessed. The intent is that this list would be written into a file, and then the CCS maintainer could add the listed words to the permanent lexicon.

The file "ccs-funcs.lisp" also contains a family of functions that handle the comment and other output of the system. All system output goes through a single function, ROUTE-OUTPUT, which allows the output options to have their effect in one place. A family of specific functions are defined, which call ROUTE-OUTPUT, and allow one to optionally specify the set of destinations possible for each piece of output. The maintainer may want to modify where specific items of information are displayed using these mechanisms.

A higher-level set of output functions includes the mechanism for writing messages. Messages can be generated by the production rules using the basic function, WRITE-MESSAGE, and the pair of functions ACCUMULATE-MESSAGES and OUTPUT-MESSAGES. WRITE-MESSAGE is used in a criticism rule to directly output a message. The message will appear each time the rule fires. ACCUMULATE-MESSAGES is used when a rule might fire multiple times, such as situations where a sentence contains multiple unresolved references. Instead of having the same message appear several times on a single sentence, ACCUMULATE-MESSAGE is used to simply accumulate any variable information from each firing. Then a later production rule invokes OUTPUT-MESSAGES to put out a single message that contains a list containing the variable information. This makes possible a much more compact, but still detailed, output.

Both of the message functions are controlled by the terse/verbose output option. The logic is that in the *verbose* mode, a long and detailed message is printed with each firing of a criticism rule. In the *terse* mode, the long form of the message is printed the first time the rule is fired, and a terse form is printed for every firing thereafter. Thus these functions keep track of whether the rule has fired previously or not. This record is cleared when a new passage criticism is begun.

The "ccs-funcs.lisp" file also contains a function that outputs the topic structure of the passage, given that the criticism rules have built a structure in passage memory that shows the topics and subtopics. PPS is supplied with a pre-defined dummy topic structure in passage memory. If the criticism rules detect topic structure, they replace the corresponding portions of the dummy structure with actual items in the text. There is also a small set of functions that print out a condensed version of the parse tree by eliminating most of the embedded network names. The idea is to provide a much more concise description of how the sentence was parsed. Finally, this file contains a set of convenience functions which the developer will find useful in rapidly iterating through revisions to the grammar or criticism rules. These are mentioned in the maintenance examples below.

2.2 The Preprocessor

The preprocessor has a straightforward but tricky job. It scans the input text file and tries to isolate individual sentences which it then passes on to the rest of the system. The rest of the system processes a sentence as a LISP list. Thus the input sentence *This is a sentence.* in a text file gets turned into the LISP expression (THIS IS A SENTENCE \.). Note that symbols that are of significance to LISP, such as periods, apostrophes, and commas, have to be set off with the COMMON LISP escape characters (backslash or vertical bars), so that these characters are treated as the print names of symbols, rather than the special functions they have in COMMON LISP. Thus the preprocessor treats punctuation specially. In addition, it recognizes text-formatting commands which are defined as having a period in position one of a line. All subsequent information on that line is ignored, but if the text-formatting command has a defined meaning to the preprocessor, the lexicon entry corresponding to that meaning is inserted as the first item in the sentence list.

The preprocessor is basically a simple parser organized around a CASE function that uses a static global variable *STATE to maintain the state of the parse. Provision is made for supplying an initial state to the preprocessor, so that control can leave the preprocessor and return, but thus far this facility has not been used. The preprocessor reads the input file defined in the global *CCS.INPUT-FILE. It uses a variety of internal streams to accumulate the characters for text-formatting commands, punctuation, and words, and assembles the result into a list which is then returned.

Sentence termination. Something to watch out for is the logic for determining whether a sentence end has been reached; this is done at the state COLLECT-WORD and in the function COLLECT-WORD. If the final character of the word is a period, then it is possible that the period does not designate the end of a sentence, but rather is part of an abbreviation. The lexicon is checked for the word and if it is an abbreviation, then processing continues. Otherwise the period is taken to mark the end of a sentence. It is clear that there may be trouble if the last word in a sentence is a bonafide abbreviation. No instances of this have been encountered in the samples, but it may be a source of future problems.

The colon character (:) is a nasty way to terminate a sentence. The current version of both the preprocessor and the grammar is probably not quite correct on the treatment of the colon. The colon normally terminates a sentence that is otherwise syntactically complete. However it can be followed by either a list of items, or another complete sentence. In response to one of the samples, the colon was here defined as a partial sentence terminator, and the grammar might accept it as being a possible end; but the preprocessor does not. Be on the lookout in the future for problems associated with colons.

Text-formatting commands. Below is shown a special function in the preprocessor that is likely to need modification depending on the materials to be processed. This is the COMMAND-TRANSLATION function.

```
(DEFUN COMMAND-TRANSLATION (COMMAND)
(LET (TRANSLATED-COMMAND)
;This is the command translation table.
;Any not listed are given a value of NIL.
(SETQ TRANSLATED-COMMAND (GETF '(
  \.TB >TITLE
  \.P0 >ORGANIZATIONAL-HEADING
  \.P1 >TOPIC-HEADING
  \.SP >PARAGRAPH
  \.BP NIL
  \.tt >TITLE
  \.mh >ORGANIZATIONAL-HEADING
  \.sh >TOPIC-HEADING
  \.pp >PARAGRAPH
) COMMAND))
(IF TRANSLATED-COMMAND TRANSLATED-COMMAND NIL)
))
```

This function translates a text-formatting command encountered in the source file into a lexical entry that can be recognized by the rest of the system. There is a simple list in property list format in which the source text-formatting command is followed by a corresponding lexical entry. Note that more that one text-formatting command can be assigned to the same lexical item; e.g., >TITLE is the interpretation of both .TB and .tt. These assignments can be expanded or modified simply by modifying the list in this function.

Other special word treatments. COLLECT-WORD also provides another function of determining whether the word appears in all capital letters, or has its first letter capitalized. The current version makes no use of the first-letter-capitalized features, but the all-letters-capitalized is taken to define an acronym, which the FIND-ACRONYM function in "lex-funcs.lisp" will use to define the word as being a noun if it is unknown. This feature was incorporated because acronyms are very frequent in Navy materials, and sometimes an acronym is used that corresponds to an existing word that does not have an otherwise compatible grammatical definition (such as ODD used as a noun instead of an adjective). This acronym defining facility is sensitive to whether the entire sentence has appeared in capitals, in which case only an unknown word will be accepted as an acronym. Of course acronyms can be added to the lexicon as standing lexical items, but this facility was added to make it unnecessary to define each possible acronym.

Another function of COLLECT-WORD is to take words that contain only numeric digits and put the special symbol of greater-than (>) at the beginning. This is to force the rest of the system to treat a number, such as 1972, as a symbol, not as an actual number.

2.3 The Lexicon

There are two lexicon files, "lexicon" and "supplementary-lexicon." The main lexicon file contains roughly 11,000 entries, and thus can take a substantial amount of time to load. The supplementary lexicon is much smaller, and thus new items or experimental items can be placed in this file, which can then be loaded into an image containing the main lexicon in a much shorter amount of time.

Main lexicon. The lexicon currently does not make use of any of the redundancy between different forms of words, such as nouns and their plurals, or different forms of verbs. Each different lexical form is represented with its own entry. Each word is represented as a LISP symbol, and is stored as LISP properties of the word. Notice that because LISP symbols are used to represent the words in the lexicon, distinctions between upper and lower case are not represented in the lexicon. The following is an excerpt from the lexicon file; this will help explain the format of the lexicon entries.

```
(|,| PUNCTUATION)
({ PUNCTUATION)
(} PUNCTUATION)
(>POSS POSS-MARK PUNCTUATION)
(A DET NDEFDET ROOT WORD)
(ABANDON VERB (ABANDON) VERB-INF ROOT WORD)
(ABANDONED VERB (ABANDON) VERB-PSP WORD)
(ABANDONING VERB (ABANDON) VERB-PRP WORD)
(ABANDONS VERB (ABANDON) VERB-3PS WORD)
(ABBREVIATE VERB (ABBREVIATE) VERB-INF ROOT WORD)
(ABBREVIATED VERB (ABBREVIATE) VERB-PSP WORD)
(ABBREVIATES VERB (ABBREVIATE) VERB-3PS WORD)
(ABBREVIATING VERB (ABBREVIATE) VERB-PRP WORD)
(ABBREVIATION NOUN (ABBREVIATION) ROOT WORD)
(ABBREVIATIONS PLUNOUN NOUN (ABBREVIATION) WORD)
(ABILITIES PLUNOUN NOUN (ABILITY) WORD)
...
STOP
```

Each lexicon entry is a single line in the lexicon file, and is enclosed in parentheses. Each entry is thus read as a single LISP object, a list. The last item in the file must be the atom STOP. The CAR of the list is the lexical item, and the remainder of the list is a condensed property list. The format of the condensed property list is that if an atomic item, such as VERB, is followed by a list, such as (ABANDON) for the entry ABANDON, then the atomic item is the property name and the list item is the property value. Otherwise the atomic item is simply a property name whose value is to be T and the next item is simply another property name. A function in the file "lex-funcs.lisp" expands these condensed property lists into full properties which are stored on the symbol for the lexical item itself.

Referring to the example, the first four items define some items of punctuation. Notice how the comma in the first item must be enclosed in vertical bars, since it is used in COMMON LISP to define a symbol whose name is that single character. The fourth item, >POSS, is an example of special items generated by the preprocessor and other preliminary processing done on a sentence prior to parsing. The greater-than sign (>) has been used as the special character to designate an artificial lexical item. In this case, >POSS is used to indicate a possessive construction; the preliminary processing functions replace apostrophes and the accompanying *s* with this symbol to normalize the form of a possessive construction.

The remaining entries in the example illustrate a few points. The property ROOT means that this is the root form of the word. For example ABBREVIATION is marked as ROOT whereas ABBREVIATIONS is not. Notice how the noun property for ABBREVIATIONS refers to the root form of the word. CCS currently does not use this fact to any great extent, but it is available in the lexicon for future use in more semantically-based processing. Also one can see how the different forms of the verb ABBREVIATE and ABANDON are all explicitly shown. VERB-INF is the infinitive form of the verb, VERB-PSP is the past participle, and VERB-PRP is the present participle. VERB-3PS is the special form for the third person singular use of the verb as in *He abandons the ship*. Notice that all of these words have the property WORD; this is to distinguish them from the punctuation items. This property is used occasionally to determine whether a word encountered in the input is known in the lexicon or not.

The lexicon file can be updated simply by editing it. A new entry can be added, preferably at the corresponding alphabetized location, or the properties modified. One should take care that the parentheses are properly balanced within each new or modified item; the symptom of unbalanced parentheses will be quite obvious when loading the file, but it will probably take an inconveniently long time to show up. The lexicon file as currently defined must end with the atom STOP. This is a vestige of earlier LISP implementations not being able to halt properly upon end-of-file.

Supplementary lexicon. The entries of the main lexicon are mostly standard English words. The supplementary lexicon has been used for various special and unusual lexicon definitions. A condensed version of the supplementary lexicon file appears below:

```
;text formatting command lexical items
(>HEADING PUNCTUATION HEADING-MARK TEXT-FORMAT-COMMAND)
..
(>PARAGRAPH PUNCTUATION HEADING-MARK TEXT-FORMAT-COMMAND)
        ;eg like run-in, or simply a paragraph start
(>END-OF-SENTENCE PUNCTUATION)
...
(>TIME-OUT PUNCTUATION)
...
;present-participle verbs used frequently as nouns
(PITTING NOUN (PITTING))  ;word is already defined
(TESTING NOUN (PITTING))  ;word is already defined

(O.D. ABBREVIATION NOUN (OUTSIDE-DIAMETER) WORD)
(I.D. ABBREVIATION NOUN (INSIDE-DIAMETER) WORD)
...
(TORCH-CUT VERB (TORCH-CUT) VERB-INF ROOT VERB-PSP TRANS WORD)
(CHISEL-CUT VERB (CHISEL-CUT) VERB-INF ROOT VERB-PSP TRANS WORD)
(WORLDWIDE ADV (WORLDWIDE) ROOT WORD)
...

(ONLY_IF IDIOM PROP-CONJ-A-B (ONLY_IF) ADVCL-CONJ (ONLY_IF) WORD)
(BECAUSE_OF IDIOM PCON-A-B (BECAUSE_OF) WORD)
(TO_THE_RIGHT_OF IDIOM PREP (TO_THE_RIGHT_OF) WORD)
(REGARDLESS_OF IDIOM PREP (REGARDLESS_OF) WORD)
(RESPONSIBLE_FOR IDIOM PREP (RESPONSIBLE_FOR) WORD)
(CONSISTS_OF IDIOM VERB (CONSISTS_OF) VERB-3PS WORD)
(AND_/_OR IDIOM CONJ (AND_/_OR) WORD)
(SUCH_AS IDIOM PREP (SUCH_AS) WORD)
(FOR_EXAMPLE IDIOM ADV (FOR_EXAMPLE) WORD)

;sample jargon
(ABOARD_SHIP IDIOM ADV (ABOARD_SHIP) ADJ (ABOARD_SHIP) WORD)

;sample job title
(|ELECTRICIAN_'_S_MATE| IDIOM NOUN (|ELECTRICIAN_'_S_MATE|) WORD)

STOP
```

A couple of items, such as >HEADING and >PARAGRAPH, are special punctuation symbols that are inserted by the preprocessor in response to encountering text-formatting commands in the input file. Note that the properties on this are both HEADING-MARK and TEXT-FORMAT-COMMAND. Both the grammar and the semantics module respond to these in useful ways.

Notice that if a source file is supplied as input that has text-formatting commands not known to the preprocessor, they will either be ignored if they correspond to the currently defined syntax for text processing commands, or will simply pass through where they may or may not be recognized as existing lexicon items or defined as new words. Thus if text-formatting commands are to be included in the input, the preprocessor must be prepared to accept them.

A few items in this excerpt, PITTING and TESTING, are defined as nouns. These words are already present in the main lexicon defined only as verbs. One of the samples used to develop CCS had these words, especially PITTING, heavily used as a noun. It would have been possible to modify the grammar so that any present participle verb would be accepted as the head noun of a noun phrase, but such a sweeping decision also had undesirable side effects in parsing other sentence forms. The most conservative strategy is to simply use the supplementary lexicon to define popular verb-based nouns such as PITTING or TESTING. If experience shows that there is a very high frequency of such words, then the grammar would need to be suitably modified and strengthened to accept all such words as nouns.

The next two items are some examples of abbreviations. Abbreviations have a special status since they contain periods, which normally mark the end of a sentence to the preprocessor. When the preprocessor encounters a period, it checks to see if the item immediately preceding the period is an abbreviation, by looking to see if the constructed word containing the period is listed as an abbreviation in the lexicon. If an actual abbreviation is not in the lexicon, the preprocessor is likely to break the abbreviation in two, most likely giving an incorrect parse. Thus, the most commonly used abbreviations should be added to the lexicon in the manner shown here.

The next few items, TORCH-CUT, CHISEL-CUT and WORLDWIDE are examples of peculiar verbs and a somewhat unusual adverb that were encountered in working with one of the NPRDC samples.

The next several items are some of the *idiomatic expressions* currently in the supplementary lexicon. An idiomatic expression is defined as a fixed sequence of words which in some sense plays exactly the same grammatical role as a single word would. A good example is the phrase *to the right of* ; this could be processed literally as a prepositional phrase embedded inside another prepositional phrase, but in fact *to the right of* defines a single semantic relationship of one object being located in a certain direction from another object, in the same kind of way as simpler prepositions such as *near* or *below*. Thus, to save the parser from doing more work than necessary, and to obtain a simpler semantic structure, it makes sense to define *to the right of* as denoting a single relationship. Thus this phrase is defined as an idiom. Prior to parsing a sentence, one of the lexicon functions in the file "lex.funcs" is used to find the idiomatic expressions. The *FIND-IDIOMS* function scans the sentence, and forms every sequence of words up to a certain maximum length (currently four) and checks to see if the resulting compound word is defined as an idiom in the lexicon. The convention for connecting the words into the compound words is to separate the original individual words with an underscore character, as shown in this excerpt. If the sequence of words is listed as an idiom in the lexicon then the sequence of words is replaced with the compound word defined as the idiom. Thus, as you can see in the excerpt from the supplementary lexicon, each idiomatic phrase is defined as a sequence of words separated with underscore characters, given the property IDIOM, and then given some syntactic properties, such as being a preposition designating the relationship TO_THE_RIGHT_OF.

My recommendation is to use idioms to define unusual or jargon phrases, or for phrases that have a very fixed conventional structure, such as those shown in the excerpt. The excerpt shows a piece of sample jargon, *aboard ship* which syntactically could be processed as a condensed prepositional phrase, but is often used in unusual places in the sentence such as both as an adverb or as an adjective. Defining this piece of jargon as an idiom simplifies processing considerably.

Another stereotyped kind of phrase is a job title such as *Electrician's Mate* or *Machinist's Mate*. Notice that the apostrophe and the *s* are separated in the idiom definition; this is the work of the preprocessor, one of whose basic functions is to separate punctuation from the adjacent words. As noted before, the last item in the supplementary lexicon file is the atom STOP.

Loading the lexicon. A lexicon file is put into effect by loading it with the function called LEXICON-LOAD in the file "lex-funcs.lisp". This function reads the definitions in a file, expands the condensed property list, and assigns the properties to the individual words. Thus, when this function has finished, every symbol listed as a lexicon entry in the file will have the listed properties on its property list. Since this is a fairly slow process for the entire lexicon, the relevant functions for loading the lexicon have been separated into a much smaller set of functions in the file "lex-build.lisp". An image can be constructed by using these functions to load the lexicon, and then the resulting image can be saved. The rest of the system can then be built relatively quickly by activating the lexicon image, loading the remaining functions of the system, and compiling the grammar and production rules.

Lexical preparation of the sentence. Several passes are made on the sentence to prepare it for parsing, which cause new lexicon items to be defined. Notice that these run-time additions to the lexicon do not modify the lexicon file; they have effect only during the execution of the current image. Thus, if more than one file is criticized within a single image execution, all of the new lexicon definitions constructed during the criticism of the first file will be in effect during the processing of the second file. Provision is made for writing the new lexicon entries into a file using the GUESSED-WORDS output option. The contents of this file can then be added to the permanent lexicon file and a new CCS image built.

The relevant function for these operations is PREPARE-SENTENCE in "ccs-funcs.lisp". This function calls several other functions which are in the file "lex-funcs.lisp". The first pass removes the marker for initial capital letters installed by the preprocessor, because this information is not currently used. The next pass finds any acronyms in the sentence and add them to the lexicon if they are not already known. The next pass through the sentence consists of finding the idioms and modifying the sentence accordingly, followed by a pass that finds and replaces apostrophes with the special possessive mark (>POSS). Once the sentence has been processed this far, the remaining words in the sentence are checked to be sure they are all known, and any unknown words are assigned a guessed part of speech.

Guessing the part of speech is done by two functions. RECOGNIZE-CLASS recognizes number strings generated by the preprocessor (e.g., >1974), or labels, which are unknown words containing numbers; these are very common in technical text (e.g., *V104*). If an unknown word is not one of these two classes, a function GUESS-WORD-CLASS is invoked. This does an analysis of the ending of the word to see if it can be assigned a grammatical class. For example a word ending in *ive* is almost certainly an adjective whereas one ending in *ly* is probably an adverb. This function attempts to also construct the root form of a guessed verb; this algorithm is slightly defective in that it does not do a proper job with verbs whose root ends in *e.* If all else fails, the word is guessed to be a noun since this quite often is a correct guess.

2.4 The Parser

Overview. The parser is perhaps the most subtle, difficult and critical part of the entire system. Analyzing the grammar of sentences is a severely difficult computational problem which has not in general been solved. This parser uses a combination of old approaches, and so is severely limited in some ways compared to more modern ideas, but the *coverage*, the variety of syntactic structures, that the parser can handle is quite large compared to many newly proposed parsers.

The philosophy of much of the grammar is to make use of constraints in English grammar to help parse the sentence. For example, the grammar tests for agreement of the subject of a verb and the form of the verb in person and number. This has the disadvantage that if the writer has made a grammatical error, the sentence will fail to parse, and the system will be able to make fewer useful comments about the sentence. On the other hand, it has the advantage that such constraints on grammatical structure help parse the sentence in a reasonable amount of time, and normally in the proper fashion.

The basic form of the parser is that it is an augmented transition network (ATN) parser. This document will not try to explain the basic principles of ATN parsers; these can be found in almost any textbook on artificial intelligence; it is one of the oldest approaches to syntax analysis. The file "parser.lisp" includes two families of functions. The first is the compiler for the high-level grammar specification file, and the second is the actual ATN interpreter. The compiler reads the grammar definition file, and constructs the ATN. The interpreter is then given a sentence and follows the ATN and returns a parse tree for the sentence.

Grammar specification. There are two relatively unusual twists to the ATN approach that are used in this parser. First, the transition network is not coded directly, but is compiled from a fairly compact linear notation, the *High-Level Grammar specification language* (HLG) described in Mayer and Kieras (1987). Thus extensions to the grammar are made by working in a compact and simple specification language, and the new grammar specification is then compiled into a transition network. Experience has shown that attempting to work with an ATN directly is impractical when it becomes as large and complex as required for this sort of material. Thus using the high-level specification language has made this fairly large parser relatively easy to develop.

The chart mechanism. The second twist on the parser is that it incorporates a *chart* that greatly speeds up the ATN processing. That is, the classic problem with an ATN parser is that if the parse fails, the ATN interpreter backs up to each previous decision point in the network and tries all possibilities at that point. Thus, if one of the earliest decisions in processing the sentence was incorrect, the ATN parser will have to back all the way up to that decision point, after exhausting all other alternatives along the way, and then reanalyze the sentence. Normally this reanalysis would be done completely from scratch. Thus, it can take an ATN parser a very long time to analyze an unusual sentence, and an extremely long time before it discovers that it cannot parse the sentence at all. This is remedied by using the chart facility as discussed somewhat in Winograd (1983) and Allen (1987).

The chart is simply a table; whenever a constituent, such as a noun phrase, is found, an entry is made in the chart that shows that a noun phrase has been found starting at a certain position in the sentence and extending for a certain length in the sentence. In the future, if the ATN requests a search for a noun phrase at that same position, the results are simply found in the chart, and then processing can continue immediately. Likewise, if the parser discovers that there is no noun phrase to be found at a certain position, an entry is made in the chart that there is no parsing path through the noun phrase network at that point. Thus, rather than waste its time in futile re-explorations, the parser simply goes on to try something else. If the parse is successful, a chart entry is eventually made that covers the entire sentence. The chart itself is rather large, but without the chart, the parser would be indulging in multiple repeated combinatorial explorations, so the chart generally results in much faster parsing. Thus this strategy represents a substantial tradeoff of space for time.

ATNs normally use *registers,* which are variables that can be assigned values during the parse which are then used later. The ATNs described in the literature typically use the registers for a variety of purposes, but here they are used only for a steering function. For example, the noun phrase network will set a register to indicate the person and number of the noun phrase; the verb phrase network will then use the register value to steer the parse to check for the corresponding form of the verb. Making use of a chart in conjunction with registers is especially tricky. The approach taken here is to save with each chart entry the values of any registers that were set during the processing of the network, and any register values that were tested during the processing of the network. A chart entry will be reused only if the current value of the tested registers agree with those in the chart, and reusing the chart entry entails setting the registers to any values that were noted as set in the chart entry.

There is also a provision for scoping register in a network. A variable scoped for a network means that the variable is defined as having a new binding within that network and all subnetworks that are dynamically called by that network. For example, in the current grammar, this scoping is used to ensure that modifiers of a noun phrase do not change the value of the register ?PERSON-NUMBER, which is then tested in the verb phrase networks to ensure proper agreement of the verb with the subject. Most of the mechanism is available for maintaining these register notations in chart entries automatically, but in the current version, the grammar programmer is responsible for specifying which registers are set and tested by individual networks and for defining the scope of registers.

Syntax of HLG grammar definitions. The definition of a grammar for the parser consists of a series of network definitions in a file. The syntax of these expressions is given below:

```
(NET-DEF <network-name> {optional register specification}
 expression-series
)
```

The NET-DEF expression is used to define an ATN network. The name of the network must start with a dollar sign ($). Following the network name there is an optional register specification and then there is an expression of the type called an expression series. The syntax of the network specification is of the form

```
:REGISTERS (type-keyword register-list  ...)

type-keyword = :SCOPED or :SET or :TESTED
register-list = (register-name  ...)
```

The significance of the specification of the register type for the network has SCOPED, SET, or TESTED is discussed more in the section explaining how the parser interpreter works. Each of these type specifications is followed by a list of register names of the specified type. A register name should appear as only one type in a network definition.

The expression series specifies a pattern of sentence constituents that must be present for parsing to succeed in this network. An expression series can consist of a single item, a sequence, or an alternation, where a sequence is a series of items that must appear in the listed order and an alternation indicates that any one of the items may appear, but only one:

```
expression-series =
     (item)
  or sequence
  or alternation

sequence = (item item item ...)
alternation = (item / item /... item)
```

Each item in an expression series can one of several types as listed below

```
item =
          <grammatical-category>
     or   <network-name>
     or   !<literal>
     or   - expression-series    ;optional (0 or 1)
     or   + expression-series    ;one or more
     or   * expression-series    ;zero or more
     or   code-expression
```

The grammatical category is the name of a grammatical category such as NOUN for a noun, or ADV for an adverb. This specifies that the current word must belong to that grammatical category. If the item is a network name, it means that at this point in the parse a subnetwork should be invoked to see if the specified constituent is present. The literal specification tests for a specific literal word or character. There are three types of items that indicate various forms of iteration or repetition of an embedded expression series. These are optional, one or more, or zero or more, instances of the structure indicated by the following expression series. One final type of item is a code-expression. These specify a test that allows a transition if the arc is crossed, where the test is either a piece of LISP code or a register operation:

```
code-expression =
    (LISP <form>) ;test is satisfied if <form> evaluates to non-NIL

 or (TESTR <register-name> <value-form>) ;test satisfied if value in <register-name> is
                                         ;EQL to value of <value-form>
 or (TESTR NOT <register-name> <value-form>) ;test satisfied if value in <register-name> is
                                             ;NOT  EQL to value of <value-form>
 or (SETR <register-name> <value-form>) ;set <register-name> to value of <value-form>
                                        ;with the test always satisfied
```

The LISP expression allows one to include an arbitrary piece of LISP code to constitute a test for whether the arc should be crossed. The arc is crossed if the specified form evaluates to non-NIL. The TESTR expression allows one to test the value of a register. The test is satisfied if the value in the named register is EQL to the evaluated value form. The TESTR NOT expression is a negation of the regular TESTR expression; the test is satisfied if the register is not EQL to the value. Finally, the SETR form is used to set a named register to a value; however, the test is always satisfied.

Here is an example of some HLG network definitions that will help explain the syntax:

```
(NET-DEF $INITIAL-VERBMOD
    (+ ((ADV / NEG / $PREPPHR) - (!\,)))
)

(NET-DEF $PREPPHR :REGISTERS (:SCOPED (?PERSON-NUMBER) :TESTED (?POST-VERB))
    (- (ADV) $PREP $NP)
)

(NET-DEF $PREP
    (PREP * ((!\, / CONJ / (!\, CONJ)) PREP -(!\,) ))
)
```

The first NET-DEF defines a network named $INITIAL-VERBMOD. This network is used to describe a constituent that modifies a verb and that can appear at the beginning of the sentence. The network specification is that we must have one or more items, each of which can be an adverb, a negation word (such as *never*), or a prepositional phrase. Each such item can be followed by an optional comma. The $PREPPHR is a reference to another network, which is defined in the second NET-DEF. This network has a scoped register and a tested register specified. As described in the section on the parser interpreter, the scoped register takes on a new value, in the same manner as a locally defined variable does in LISP; the previous value is restored when we leave this network. The tested register means that later on, either in this network or in some subnetwork, this register will be tested for the specific value. The tested register is specified here to inform the parser interpreter that this value should be saved in the chart. A prepositional phrase is defined in this network very simply as an optional adverb followed by a preposition (which is specified by yet another network), followed by a noun phrase which is also defined in another network. The nature of a preposition is defined in the network $PREP, which shows that a preposition form can consist of a single preposition (grammatical category PREP), followed by zero or more sequences in which commas or conjunctions both can appear, followed by another preposition, followed by an optional comma. This network allows for compound prepositions such as *above and below* to play the role of a single preposition.

The HLG compiler. The compiler has a relatively simple structure. The top-level function reads a network definition from the input file; the syntax of this definition file is such that each network definition is a single top-level object in the file. The network definition is handed to a series of functions, basically one for each type of construction in the network definition. Each of these functions is given as input a START and END network node, and returns updated START and END nodes. These are the starting and ending nodes for the piece of the parsing network that each function builds. As described in the Mayer and Kieras (1987) report, there are some inefficiencies in how this network is constructed, which a future revision may attempt to optimize. The syntax for defining registers, and setting and testing them, is new to this version of the system and not described in the Mayer and Kieras report.

The network data structure consists of the following: Each node in the ATN network is a symbol; the first node in each network is simply the name of the network, and the names of nodes within the network are simply the network name with a numeric value appended. The actual contents of the network can be examined by looking at the values for the network name and then looking at the values for the various other node names defined in the network. Each node is defined in the value of the symbol as simply a list of arcs, where an arc is a structure that includes the type of the arc, the value, and the name of the next node. The type designates what test should be performed on this arc and the value is information used in the test. Some specifics are given in the maintenance examples later in this report.

ATN interpreter. The interpreter function is large and complex. At the top level it consists of a series of functions most of which are called from only one place. The code was prepared in this fashion to make future modifications and extensions easier, but there is clearly some processing time being lost in calling from one function to the next. If your LISP implementation supports the compiler INLINE directive, some speed improvements might be obtained. I suggest modifying this code only extremely carefully; it is very hard to debug, since analyzing a sentence typically involves many thousands of function calls.

The basic philosophy of the ATN interpreter is to keep a stack of arcs. The basic processing is to take an arc off the top of the stack and perform the test associated with it. If this takes us to a next node in the network, the first arc for that network will be pushed onto the stack, and then we will then loop around to the top of the stack again. Now to augment this simple picture: The arc on top of the stack is not really the next arc to be followed, but is an arc frame containing a number showing the last arc that was used. To process an arc, we increment this number, which gives the next arc leaving the node. If no arcs have yet been followed out of a node, this number is negative, so the first arc we try is numbered zero. Then as a further complication, the stack actually contains two kinds of frames. The first is an arc frame, which is as has been described. The second is a net frame, which is a record of when we have called a subnetwork. The net frame records information about the invoking arc, which is the arc that invoked the subnetwork.

If we fail to leave a node, we pop the previous frame off the stack. If it is an arc frame, we simply follow the next arc leaving the earlier node and thus try the next previous decision point. However, if it is a net frame, this means that we have failed to find a path through an invoked network. We return to the node containing the invoking arc and try the next arc from that node.

But if we process a POP arc, it means that we have succeeded in finding a pathway through the network. The chain of frames since we called the network is taken out of the stack and a new chart entry is made that contains this pathway. Thus the chart entry represents the fact that we found a constituent corresponding to the network, and the pathway in the chart entry shows the chain of arcs or network calls that were used to find that constituent.

If we fail to find a pathway, as indicated by backing up to the frame in the stack where the network was called, an entry is made in the chart that there is no pathway for that network at this position in the sentence.

If a network is called from an arc, the conventional thing for an ATN to do would be to start processing the first arc leaving the first node in that network. However, making use of the chart entails first checking the chart to see if there is an entry for that network leaving this position. If a qualifying entry is found, the net frame for that chart entry is simply placed on the stack, and we continue at the point in the sentence where the chart entry says the network has reached. If we fail to parse from this point, we will not give up on this network until we have exhausted all of the chart entries for that network, and then if more paths are possible, we will back up into the network to look for a new parse. Some of the more subtle code in the parser is ensuring that the chart never gets redundant entries, because these slow the processing considerably.

The parsing stops when there is a POP out of a special, top-level network defined with the reserved name $START. This network must be included in the grammar definition. When we pop from $START, the parse is complete. The stack at this point will contain a single net frame which refers to the chart entry for the top level constituent in the sentence.

The parse tree returned from the parser is computed by the function STACK-TO-TREE. This simply unpacks the stack using the chart entries to recover the parsing path through the sentence in terms of the networks and arcs. The path of arcs is examined for the tests to determine the grammatical category used to account for each word in the sentence. The parse tree that results is a nested list, where the CAR of each list is either the name of a network or a grammatical category for a word. It is this tree that is listed as the detailed output if so requested. Another function, PARSE-PRINT in "ccs-funcs.lisp" prints a simplified form of this tree by stripping out all of the lower-level network names. Here is an example parse tree produced for the sentence
The large motor on the engine always powers the pump.

```
($START
 ($STATEMENT
  ($STATEMENT-SIMPLE
   ($DECLARATIVE-STATEMENT
    ($NP
     ($NP-SIMPLE (DEFDET THE) ($ADJ ($ADJ-SIMPLE (ADJ LARGE))) (NOUN MOTOR)
                 ($RELCLAUSE
                  ($SUBRELCLS
                   ($PREPPHR ($PREP (PREP ON))
                             ($NP ($NP-SIMPLE (DEFDET THE) (NOUN ENGINE))))))))
    ($CLSPRED
     ($CLSPRED-SIMPLE
      ($VERBPHR
       ($VERBPHR-SIMPLE
        ($VERBPHR-ACTIVE
         ($VERBSEQ-ACTIVE ($VERBMOD (ADV ALWAYS))
          ($VERB-SIMPLE (VERB-3PS POWERS)))
         ($VCOMP-ACTIVE ($NP ($NP-SIMPLE (DEFDET THE) (NOUN PUMP))))))))))
  \.)
 >END-OF-SENTENCE)
```

Non-parsable input. The parser is actually called by the function ATTEMPT-PARSE in "ccs-funcs.lisp". This function is part of the strategy for dealing with non-parsable input. The grammar contains a network $GRAMMAR-PROBLEM that is routinely resorted to if no other top-level analyses of the sentence succeed. $GRAMMAR-PROBLEM is simply another parsing network, and attempts to analyze the sentence just in terms of miscellaneous noun phrases, punctuation, and so forth. However, sometimes non-parsable sentences cannot be analyzed even by $GRAMMAR-PROBLEM, or it is possible for sentences to involve structures so deeply nested that the combinatorics of possible parses are excessively large. For this case, the parser includes a time-out variable, PARSER:*TRANSITION-COUNT-LIMIT, that halts the parser if the specified number of transitions (arc crossings) in the network have been exceeded. In this case the ATN interpreter returns a special value which ATTEMPT-PARSE detects, and then the parser is called again with a special symbol >TIME-OUT at the beginning of the sentence. The current grammar then calls another special network, $SENTENCE-TOO-COMPLEX, which attempts an even simpler analysis of the sentence. Part of the logic is in place for reusing the chart after the first failure, so that this reparse will be considerably faster. However this piece of work was not completed. If there are many cases of time-out failures in actual materials, it might help to finish implementing this feature.

A technical note about the parser: The data structures used in the parser were chosen after some benchmark tests on LUCID COMMON LISP. It was found that the property list format, accessed by the functions GETF and SETF, were extremely fast. Thus, the chart is simply a doubly embedded list, where a GETF using the network name followed by a GETF using the current position could recover a specific chart entry quite rapidly. It is possible that newer or different LISP implementations would dictate other choices. But notice that array structures are generally not applicable simply because there are variable number of entries, and no clear maximum number, for most of the situations where one would be tempted to use an array.

2.5 The Semantics Module

Overview. The semantics module is a collection of functions in the file "propositions.lisp". The top level function is STS.TRANSLATE. This function translates a parse tree, as described above, into ACT structures. It does this by calling a series of functions, one function for each network defined in the grammar. Each function analyzes the contents of the parse tree at its level, and constructs a series of ACT links to represent the semantics content of the corresponding portion of the parse. For example, the parse tree for a noun phrase would be analyzed with a series of functions which construct the semantic network links to show that a referent has the predicates of being *large* and being a *motor*. These functions are basically arbitrary, and so are not easily summarized. However they have a basically simple structure, and are further simplified by the use of some macros defined in the "propositions.lisp" file. The example below of adding new parses and criticisms describes these macros in more detail.

Syntax of ACT structure. The semantic content of sentences is represented using a form of the ACT semantic network structure proposed by Anderson (1976). This document is not the place to discuss knowledge representation or semantic knowledge in general; suffice it to say that this representation was chosen mainly because of its great technical simplicity, evidence showing that it is at least a plausible representation for human memory of sentence meanings, and the interesting property that it is close enough to the syntactic structure of sentences that it is easy to establish a relationship between the semantic content and the original syntactic form in which it was expressed. That is, there is a semantic representation node that can be "tagged" for almost any original syntactic form. For example, there is a distinct node that represents the main proposition of a sentence, and other nodes correspond to each subordinate clause, each modification of a noun, each modification of the main proposition, and each modification of the relationship between constituents (such as prepositions). One reason why the ACT representation has this property is that it is relatively "close" to the surface form of the sentence. Other knowledge representation proposals, such as conceptual dependency, are relatively distant from the surface form of the sentences.

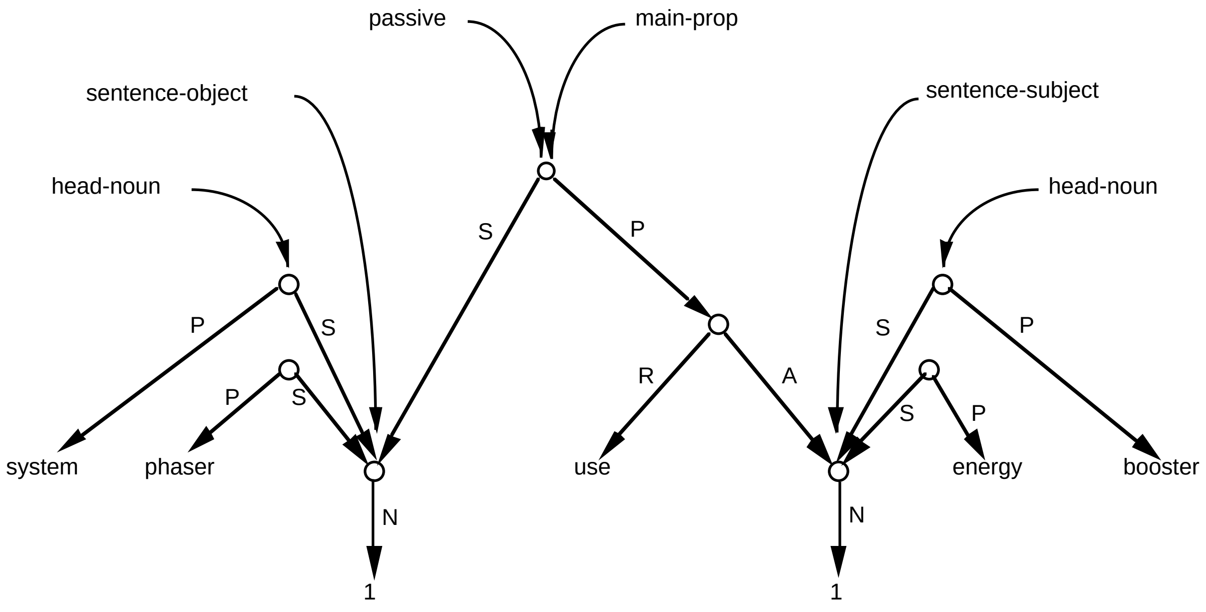

Figure 2. Example of ACT semantic network structure for the content of the sentence *The energy booster is used by the phaser system.* The arrows labelled with *S, P, R, A,* and *N* are links in the ACT structure, while the curved arrows represent *TAGS* specifying surface-structure information about the parts of the semantic structure.

A conventional way to display ACT structure is in the form of a diagram (see Figure 2 for an example). CCS operates on ACT structure in terms of a list of binary relations or *links*. Each link consists of a list containing a link type followed by a source node and a target node. The possible link types are ***S****ubject*, ***P****redicate*, ***R****elation*, ***A****rgument*, and ***N****umber*. A node is simply a LISP symbol. In CCS there are effectively three different types of nodes, and they are indicated by convention with symbols with particular letters. To these initial letters are appended digits so as to produce unique symbols. The three types are: *proposition* nodes (PROPn), *relation* or *predicate* nodes (PREDn), and *referent* nodes (REFn). The referent nodes represent the entities or concrete objects discussed in the text; proposition nodes represent propositions indicating that some referent stands in a relationship to a specified predicate. A proposition node has a subject link to the subject and a predicate link to the predicate. A predicate can be either a simple predicate concept, such as an adjective, or a predicate (relation) node. A relation node has a relation link to a relation concept such as a verb, and an argument link to an argument, such as the object of the verb. Note that the subject of a proposition need not be only a referent node; it may be another proposition node (e.g., to indicate that a proposition is false), or a relation node (e.g., to modify a verb with an adverb). The number link (N) indicates the number of items in the set represented by the referent. The following summarizes the syntax of these expressions succinctly:

```
link = (<link-type> <source> <target>)

link-type =  S or P or R or A or N
<source> = proposition node or relation node
<target> = proposition node, relation node, or referent node

Possible combinations:
(S <proposition-node> <subject>)
(P <proposition-node> <predicate>)

<subject> = referent node, proposition node, or relation node
<predicate> = simple predicate concept or relation node

(R <relation-node> <relation>)
(A <relation-node> <argument>)
<relation> = a relation concept
<argument> = referent node, proposition node, or relation node

(N <referent node> <cardinality>)
<cardinality> = a symbol for the number of members of the referent set
```

The following is an example of ACT structure resulting from the analysis of the sentence *The large motor on the engine always powers the pump.*

```
((S PROP1 REF1) (P PROP1 LARGE) (S PROP2 REF1) (P PROP2 MOTOR)
 (S PROP3 REF2) (P PROP3 ENGINE)
 (S PROP4 REF1) (P PROP4 PRED1) (R PRED1 ON) (A PRED1 REF2)
 (S PROP5 REF3) (P PROP5 PUMP)
 (S PROP6 REF1) (P PROP6 PRED2) (R PRED2 POWER) (A PRED2 REF3)
 (S PROP7 PRED2) (P PROP7 ALWAYS))
```

The proposition PROP1 specifies that the object REF1 has the predicates of *large* and *motor;* in other words, REF1 is *a large motor.* PROP3 specifies that REF2 is *an engine.* PROP4 specifies that REF1 stands in an ON relationship to the argument REF2; in other words, *the motor is on the engine.* PROP5 defines REF3 as being *a pump*, and PROP6 specifies that *the motor* has a POWER relationship to *a pump*, REF3. Finally, PROP7 applies an ALWAYS predicate to the POWER-REF3 relationship.

Operation of the Semantics Module. The basic philosophy underlying the combination of the parser and the semantics module is to separate the computational problem of correctly parsing the sentence, which is quite intimidating, from the problem of generating a semantic interpretation of the parse tree. That is, more conventional uses of ATNs involves constructing the semantic structure by means of various operations during the parse itself. However, this parser does not build a semantic structure; it simply generates a parse tree of the constituents defined by the networks. Thus for example, an active and passive sentence will have different parse trees reflecting whether the passive verb phrase network was used rather than the active verb phrase network. The semantics module will then construct semantic interpretations of these parse trees which, for example, will be essentially identical for passive and active paraphrases. The functions in the semantics module can be written in an essentially deterministic fashion, because the parser has already done all of the analyses to arrive at the final parse tree.

Thus the semantics module only has to unpack the parse tree layer-by-layer, and build a corresponding semantic structure. This is sometimes complicated by the fact that some embedded structure must be built before the final structure. For example, we don't know where to attach the proposition for an adverb until we know the name of the relation node constructed for the verb. Thus a common pattern that appears in the semantics functions is to accumulate certain portions of information from the parse tree, then call other functions to build the structure for other portions of the tree. These lower-level functions return the names of various nodes, and then the calling function attaches additional structure to these returned nodes.

2.6 The Reference Module

Simple reference. The key feature of connected text is that each sentence is supposed to refer to objects or entities mentioned in previous sentences. The analysis of text, both from the information processing point of view, and human comprehension behavior, has received considerable attention since the seventies in cognitive psychology. See Kieras (1977; 1983) for discussion of these issues in the context of the simulation model that was the precursor to CCS.

Many of the important functions of CCS are based on analyzing these *referential* relationships. Computing the referent of a reference is called *resolving* the reference. The basic assumption underlying CCS is that if these referential relationships are very simple to compute, then the text will be *coherent* and easy to understand. The reference module computes *simple reference*, which represents a particular set of ideas about what reference forms are easy to compute. Namely, a reference should either be explicitly to a *new* referent, an object not previously mentioned, or explicitly to an *old* , or *given* referent, an object that was previously mentioned in the passage. If the object was previously mentioned, it should be referred to either with the exact same word string as was previously used to describe it, or by a set of predicates (in the ACT sense) that form a *subset* of the predicates that have been attached to the object thus far. For example, an object might be initially introduced as *an auxiliary lube oil pump*. Later on, it could be referred to appropriately either by *the auxiliary lube oil pump* or as *the auxiliary pump*, which is a subset of the original description. Notice that if the text contained *the main pump* it would appear that the text was referring to a different pump. Likewise if the text simply said *the pump* it could either be the *auxiliary lube oil pump* or perhaps some other pump; the reference might easily be ambiguous.

Of course, more complicated references are possible. If *the engine* is first introduced with the sentence *the ship is powered by an engine*, later in the text it may be referred to as *the engine that powers the ship.* Although this is a substantial paraphrase from the original description of the engine, once the initial sentence has been translated into ACT structure, it has a form that is identical with the description *the engine that powers the ship*. The reference module is able to resolve references as complicated or more complicated than this. Even in this case, the basic form of the reference is that the text is still referring to an object with a subset of its previous description. If the reference module can not resolve a reference this way, it is a matter for the criticism rules to comment upon; there is a good chance that the writer of the material has assumed knowledge on the part of the reader, or has simply used inconsistent terminology or otherwise indulged in some form of incoherence.

Resolving SM references against PM. A basic step that is done in processing the propositions in CCS is that the propositions derived from the current sentence are stored in PPS in *sentence memory* (i.e., a tag of SM). After a sentence has been completely processed, its propositions are moved from SM into *passage memory* (PM). Thus as CCS processes a passage, the contents of PM grow larger and larger.

Resolving the reference involves taking the noun phrase propositions that are currently in SM and comparing them to the propositions in PM, to determine whether there is a referent node in PM whose propositions match the referent node in SM. If so, then the conclusion is that the referent node in SM is actually the referent node in PM, and so the matched propositions in SM can be discarded, and the name of the PM referent node can be substituted for the SM referent node in all of the remaining SM propositions. Thus, resolving the references consists of rewriting the SM semantic structure until the SM propositions refer to referents already defined in PM as much as possible. Thus, suppose that all of the noun phrases in the sentence *the auxiliary lube oil pump is powered by the auxiliary lube oil pump motor* are already defined in passage memory. If so, there will only be one new proposition of information added by this sentence, namely that the described motor powers the described pump.

Resolving noun phrases. The references in the form of an ordinary noun phrase are resolved as follows: When the representation of a new referent is created, the head noun is retained as the *head noun form* (HNF), and the *simple referential form* (SRF) is also retained. The head noun form defines the basic category of the referent, such as *motor* for the noun phrase *auxiliary lube oil pump motor.* The head noun form corresponds to the basic kind of an object that is being referred to. The simple referential form is available if the object is described as a sequence of adjectives and nouns, with no relative clauses or complicated modifiers. This is intended to capture the heuristic that in technical prose, objects are often referred to with a standardized string of words. It is probably advantageous for the reader if the same object is always referred to with the same string. Thus maintaining this string of words is a simple way both to determine whether the writer has followed a pattern of consistent terminology, and also as a way to easily resolve the reference.

The noun phrase semantics function calls the reference module. The first step in the reference resolution is simply to determine whether the SRF for the referring noun phrase matches the SRF of some previously mentioned referent. If so, the referring noun phrase propositions are simply dropped from further consideration, and the referent node name for the matching previous referent node name is substituted into the remaining propositions for the sentence, replacing the matched referent. If the SRF is not available, or does not match a previous referent SRF, a more tedious search of the previously mentioned referents is made, using the externally available copy of the PPS database stored on the symbol PPS:*DBCONTENTS.

The first step in the search is to find all candidate referents. These will be referents whose HNF matches the current noun phrase HNF. Then the candidates are examined to see which ones have predicates in the database that match each predicate in the referring noun phrase from the current sentence. Any candidates that fail to match on a predicate are dropped. If there is at least one candidate remaining after all predicates have been matched, then the reference has been resolved. If there are no candidates left, then the reference is unresolved, and the reference module returns all of the original propositions along with other indicators that the criticism rules will use to comment upon the apparent lack of coherence.

Pronoun references. The reference module is also responsible for resolving pronouns. Pronouns are surprisingly complicated but also very well behaved. Since this function is pretty straightforward, it will not be summarized in any detail. First and second-person pronouns are resolved to the pre-defined referents THE_WRITER and THE_READER. Reflexive pronouns (e.g., *himself*) are resolved to the referent occupying the role of the sentence subject. Third-person pronouns have a more complicated set of resolution rules. Since CCS does not make any use of semantic or general knowledge, it uses only some relatively knowledge-free heuristics for determining the probable antecedents of pronouns. The related criticism rules merely comment on what the assumed antecedent is; CCS has no way to know what it is actually supposed to be. The heuristics used are based on various psycholinguistic research that shows, for example, that a pronoun normally should refer to the subject of the previous sentence. The reference module has mechanisms for making use of gender in resolving pronouns, but this is often limited to only two - a combination of male and female versus neuter. If there is no previous antecedent that appeared as the subject of an earlier sentence, then antecedents that appeared as main objects of the previous sentences are entertained. A special type of antecedent is tracked for pronouns such as *this* ; these are assumed to refer to the previous sentence main proposition.

Accuracy of semantic representation and reference. Knowledge representation, even in the form of an idea as old as semantic networks, is a very difficult technical problem with no agreed-upon solutions. In extending or maintaining CCS, one can easily get bogged down in the niceties of exactly how various semantic contents should be represented. I have followed a strategy of trying to avoid this pitfall, based on the fact that as far as CCS is concerned, the only important features of the semantic structure are those that relate to reference, and the various criticisms that might be made. Thus, one needs to ensure that two syntactic expressions that are supposed to refer to the same object result in ACT structure that can be matched by the reference module, or that represents the information required by the criticism rules one is interested in. Whether it is technically correct in other aspects will not make much difference to CCS. An example of this can be given.

A major puzzle in computational linguistics is the *prepositional phrase attachment problem.* The problem is that a prepositional phrase that directly follows a noun that appears after a verb is ambiguous in terms of what it is intended to modify. The favorite classic example is *I saw the man on the hill with the telescope.* Did the action of seeing happen on the hill, or was it the man who was on the hill? Is it the man who had the telescope? Or did I do the seeing with the telescope? In general, this problem can not be solved without full access to not only general knowledge about the world, but also knowledge of the specific situation under discussion.

This problem turned out to be significant for CCS because a prepositional phrase directly following a noun was assumed to be a condensed relative clause modifying the noun. But if instead the prepositional phrase modifies the previous verb, the reference module would fail to find a match for the noun as modified by the prepositional phrase. The criticism rules would then generate a bogus criticism that there was no previous referent for the noun. Thus the semantic interpretation for this syntactic structure sometimes had a serious impact on the validity of the criticisms.

The current form of CCS attempts a partial resolution of this problem. Such prepositional phrases are parsed by a separate network, $PVPNPPHR for *post-verb-post-noun-prepositional-phrase*. The noun phrase function in the semantics module first attempts to resolve the noun phrase reference by treating this prepositional phrase as a modifier of the noun. If the result successfully resolves to a previously mentioned referent, then we conclude that the prepositional phrase must indeed modify the noun. If not, a search is done using the reference without the prepositional phrase. If it successfully resolves this reference, then we conclude that the prepositional phrase must be a modification of the main verb, and so the content of the prepositional phrase is returned through the nested semantics functions until the verb phrase semantics function can add the prepositional phrase as a modifier to the verb.

2.7 The Production System Interpreter

The production system interpreter consists of a set of production rules and a database. The interpreter tests the conditions of the production rules to determine if there are items in the database that match the conditions. If a production rule has all of its conditions matched then the interpreter performs the actions listed in the production rule.

The production system interpreter is the current version of the *Parsimonious Production System* interpreter (PPS) described in Covrigaru and Kieras (1987). PPS is probably the simplest production system interpreter available in terms of the syntax and generality of the production rules. However, internally it is pretty complicated, since it uses a variation on Forgy's (1982) *rete match* algorithm for doing pattern matching. Future workers may want to consider replacing PPS with some more common or generally known production system interpreter such as OPS, but this work would have to be done fairly carefully, and it is not clear whether there would be any great advantage in doing so. An important fact for such work is that the database of the production system interpreter is also used by the reference module and some of the output functions, such as those that print out topic structure, or those that fetch the referential forms of a referent. These functions assume that the database is available and has a certain format, and so would also need to be changed if the production system interpreter was changed.

The syntax of PPS rules is intended to be as simple as possible, reflecting the author's experience with the difficulty of programming large cognitive simulations in existing production rule languages. The syntax of PPS rules will be described, along with an overview of the PPS code itself. This is a fairly stable piece of code, having undergone only a few small corrections and modifications for some years now.

Syntax of production rules. The PPS database contains clauses:

```
clause = (constant_symbol constant_symbol constant_symbol ...)
```

A clause can contain a item that is a list instead of a constant symbol, but it is not possible to write PPS conditions to test individual items in the list; it can only be matched as a whole to a variable or wild card. By convention, the first item in a clause is a "tag" for the "type" of item, such as SM, PM, STEP, but PPS does not give this item any special status except in displaying the contents of the database.

The syntax of an individual rule is:

```
(<name> IF <condition> THEN <action>)
```

Each production rule has a name and then the special symbols IF and THEN marking the condition and action. The entire rule is enclosed in parentheses. The condition is simply a list of patterns which are implicitly conjoined:

```
condition = (pattern pattern pattern ...)
```

All of the patterns have to be satisfied before the entire condition is satisfied. A pattern consists of a list of items, or a list of items surrounded by a list headed with the symbol NOT, or a predicate pattern:

```
 pattern = (item item item ...) or
(NOT (item item item ...) (item item item ...) ...) or
(<predicate> item item)
```

The NOT is *not* the standard LISP function. Rather this indicates that this pattern is satisfied if the internal pattern is *not* present. These negated patterns look just like normal patterns, except they may not in turn include a NOT pattern.

The items within a pattern are either a constant symbol, a variable pattern name, or a wild card:

```
item = constant_symbol, pattern variable name, or wild card ("???")
```

The variable pattern names are prefixed with a question mark and have a scope corresponding to the production rule; that is their values are not defined outside the production rule in which they appear. Note that the same pattern variable name can be used in more than one rule, and the values within a rule are kept separate. Variables used within a NOT pattern have a special status. Such a variable must either be mentioned elsewhere in the pattern, or at least twice within the NOT to tie separate negated patterns together. If one is tempted to use a variable which does not follow these rules then one probably should be using a wild card instead.

The final type of pattern is a predicate; these test one or two variables for either equality or some other relationship or specified values: The current predicates are DIFFERENT, UNIQUE, EQUAL, and GREATER. The EQUAL pattern predicate is satisfied if its arguments have equal values. The GREATER predicate is satisfied if the first argument has a numerical value greater than the second. The DIFFERENT predicate is satisfied if the two arguments have different values. The UNIQUE predicate is somewhat peculiar but extremely useful. It is always satisfied if the rest of the condition is satisfied, but only permits a unique set of bindings for the two named variables. This will be discussed more in the example below.

The actions of the production rules are executed once for each set of variable values that satisfies the condition. The action consists simply of a list of action forms which are either functions built in to PPS, such as ADDDB and DELDB which add or delete a clause from the database, or a user-defined function. The arguments to these functions can be any normal lisp arguments, pattern variable names, or action local variable names (prefixed with "^") which are scoped within the action. These local variables are quite useful, for example, to pass values from one action function to another.

```
action = (action-form action-form action-form ...)

action-form = (ADDDB clause) or (DELDB clause) or
              (<user-defined function> argument argument ...)
```

User-defined functions must return specified values as follows:

```
  NIL - normally should be returned
  <string> - halts the interpreter, returning <string>
```

```
    (addlist deletelist) - addlist is a list of clauses to be added to the database,
                           deletelist is clauses to be deleted from the database
```

Overview of PPS. The basic concept of PPS can be found in the Covrigaru and Kieras report; here it will be described only very briefly. The essence of the rete match algorithm is to compile a set of production rules into a data-flow network. The conditions of the rules are used to construct the network; the subpatterns in the conditions correspond to nodes in the network, and the final output nodes correspond to the individual production rules. When an individual item is added to or removed from the database, it is passed into the data-flow network, and the relevant nodes are updated. Adding an item to the database may result in the node for a rule being updated, and so that rule may be fired, or cease to fire.

For a large set of rules, the data-flow network is fairly large and complex. But the rete match principle is to save as much computation as possible from one database updating to the next. That is, simple ways of implementing a production rule interpreter involve recomputing all matches every time the database is modified. However, the rete network maintains the current state of the database in terms of the patterns that are already matched. Modifying the database involves updating the states only of the particular patterns that are affected by the changed item.

Each node in the rete match network is either off or on. If it is on, it means that items exist in the database corresponding to that node. If the pattern includes variables, then a node that is on will also have an associated list of the bindings of those variables that satisfy the pattern. When the production system is first started, the nodes are all in an "empty" and off state. When a node changes state, change information is sent to all of the successor nodes in the network and those nodes are correspondingly updated. Whether or not a production rule fires depends on whether or not the node corresponding to a rule gets updated to an on or off state.

When a production rule fires it is fired with all instantiations of the variable bindings. That is, if there are five ways the variables in the pattern can be assigned to make the condition true, the action of the production rule is executed once for each binding. Unlike some other production system interpreters, PPS has no *data refractoriness* or *conflict resolution*. If a production rule has its conditions met, it will fire on that cycle regardless of whether it fired before or regardless of which other rules have their conditions met. The production rule programmer must ensure that a rule does not fire when it is not supposed to. The examples given below show how this can be done. But this decision was made in PPS because it was my experience that programming in more complex production systems was actually harder rather than easier.

The PPS code is commented extremely heavily. To a great extent, it has been written so as to execute as fast as possible, but there are some known modifications which will eventually be made to improve the running speed. The "pps.lisp" file contains two top-level functions. One is the compiler function COMPILE-PS-FILE, and the other is the interpreter function INTERPRET-PS.

The PPS compiler. The compiler function is given a file name and rule set name. The rule set name is simply appended to the beginning of each rule name, and makes it possible to have different sets of production rules defined at the same time with no name conflicts. This function calls various functions to first detect any obvious syntax errors, such as duplicate names, in the production rules. However there are some potentially nasty syntax errors that are not detected. Then other functions construct the data-flow network.

In outline, the PPS compiler works as follows: First the patterns in all of the production rules are used to build a discrimination network for clauses that appear in the conditions. This results in a set of PATTERN nodes, one for each of the possible patterns that appear in the set of production rules. The predicate patterns are not treated in this fashion; they are attached to the rule node created for each rule. They take effect only after the rest of the pattern has been matched. After constructing the discrimination net and the pattern nodes, the compiler then goes through the production rules from first to last, and examines the first two patterns in the first production rule. It forms a *combining node* for these two patterns; usually an AND node, that, in a sense, performs a logical AND between these two patterns. That is, if both patterns are present in the database, the two PATTERN nodes will be ON, and the AND node will then detect that both patterns are present, and turn ON. The compiler will then look at the next pattern and form another AND node to combine this pattern with the previous pair of patterns. The compiler continues in this fashion through the entire condition, and then goes on to the next production rule. However, before forming a new combining node, the compiler checks to see if those two patterns (or combinations of patterns) have already been formed. If so it replaces the patterns with the name of the already-existing combining node. In this fashion, the compiler uses the network that has already been constructed, subject to the constraint that it processes the patterns in the order that they are listed in the production rules. This results in a relatively fast compiling process compared to the alternatives, but the running speed depends heavily on whether the patterns were listed in an optimal order. Notice that one pattern being slightly different will cause the compiler to build additional network for the remainder of the production rule condition even if it overlaps heavily with preexisting patterns. Future revisions to PPS might include adding optimizers to improve the speed.

Patterns containing NOT are handled fairly differently. A negated pattern results in the creation of a NEGATION combining node, which is asymmetrical. A NEGATION node has two inputs; one is the positive input corresponding to the rest of the condition, the other is the negative input from the negated pattern. The negated pattern may either be a single pattern node, or an AND node that combines several patterns together. The negation node performs the analog of the function POSITIVE INPUT & ~NEGATIVE INPUT. That is, it is ON only if the positive input is ON and the negative input is OFF.

These combining nodes have been described as if they were simple logical operators. However if the patterns involved have variables, the calculations done for a node are in terms of the variables. The notion is that an AND node is ON only if the two inputs have variables that have consistent bindings relative to a list of specified variables. Likewise a NEGATION node subtracts the bindings in the negative input from the bindings supplied by the positive input, over a set of specified variables, and is ON only if there are some bindings remaining. This process is described in more detail in the Covrigaru and Kieras report and also in the very heavy comments in the PPS source file.

The combining network eventually terminates in RULE nodes, which are nodes for the the individual rules. A RULE node is ON if the production rule is to be fired in this cycle, and OFF if not. The bindings arriving at the RULE node are passed through any predicate patterns that have been specified and if any bindings are left over, then the rule node is ON; otherwise OFF.

The PPS interpreter. Each cycle of the production rule interpreter consists of first finding out which production rules have their conditions satisfied, and then executing their actions. First, all of the clauses to be deleted from the database are processed, and then all of the clauses to be added to the database are processed. This results in the updating of the RULE nodes. The production rules that are now to be fired are then known, as so the actions of each rule are executed. This results in a new set of clauses to be deleted from or added to the database and the process is repeated. The production rule interpreter halts if no production rules are fired.

The updating of the database is done by passing each individual clause into the rete-match network. Each clause is sorted to the pattern node at the bottom of the discrimination network. If the pattern node changes state, the successor nodes, which are either AND nodes or NEGATION nodes, are then updated. If these combining nodes change state, then their successors are updated, and so forth, until we may update a RULE node. This process is repeated for each clause. When the production actions are executed, the list of bindings are used in a COMMON LISP PROGV function to dynamically scope and bind the pattern variables and any local action variables, and sequentially evaluate the actions.

PPS outputs and displays. Action functions currently in CCS produce output messages, such as criticisms to the output file. Other functions can be readily defined, but be sure to observe the conventions described above for what action functions must return. In the context of CCS most of these action functions should return NIL, but if you want to update the database with such a function, you need to return the list consisting of the add list and delete list of clauses.

In a *verbose trace,* PPS shows on each cycle the current contents of the database, showing those clauses whose first item is a member of the TAGS-TO-TRACE list supplied as an argument to INTERPRET-PS. It then lists which rules have fired, and shows the variable bindings for each rule, and lists the actions performed by the rule. Using this verbose trace one can diagnose exactly why a rule does or does not fire. A *terse trace* simply lists the rules that were fired or each cycle. It is possible to control the interpreter so that it runs step-by-step, or halts after running a certain number of cycles and so forth, but these are rarely necessary to use.

Some important control variables are PPS:*TOP-LEVEL OUTPUT, which causes various internal PPS messages to appear in your top-level LISP window. PPS:*TOP-LEVEL-ALL-OUTPUT will send all output to the top-level window, both PPS messages and the contents of the terse and verbose traces. PPS:*VERBOSE-TRACE and PPS:*TERSE-TRACE turn on and off the verbose and terse traces. In my implementation, the verbose and terse traces can each be written to a separate Apollo window; if you have implemented the window functions on your platform, there are various variables that would hold the window name for the different traces.

One last output facility is the variable PPS:*DBCONTENTS. This symbol is used to store the property list copy of the database, to make it available outside of the PPS interpreter. This requires some explanation. Within the rete match framework, the contents of the database is actually stored implicitly as the contents of all of the PATTERN nodes in the network. However if code outside of the PPS interpreter wants to know what is in the database independently of the production rule matching mechanism, it needs to determine which clause patterns are present. One way to do this would be to examine all of the PATTERN nodes in the network; however this would be relatively slow. In addition, one might want to have clauses in the database that do not have PATTERN nodes. That is, a PATTERN node only exists if that pattern appeared in a production rule condition. If a production rule action adds something to the database that no other production rule tests in its condition, there will not be a PATTERN node that matches that clause. PPS will simply print out a message that the clause was not matched. But it is sometimes useful to add things to the database even if they are not going to be tested later, such as recording some conclusion of interest to the user.

For all of these reasons, there is a separate copy of the database maintained as the property list of the symbol PPS:*DBCONTENTS. When a clause is added to or deleted from the database the property list of this symbol is updated. The first item of the clause (the "tag") is used as the property name. This copy of the database is used in the debugging display of the verbose trace, and is also accessed by the reference module. The apparent inefficiency of two copies of the database seems to be justified by the greater processing speed; a search of a property list is much faster than examining all of the individual PATTERN nodes. However, future implementations may want to reconsider this decision.

3. MAINTENANCE EXAMPLES

3.1 An Introductory Example: Elided Verb Phrase Problem

This introductory example shows a small and single correction to the grammar. Technical details of the process appear in later examples. The presenting symptom of this problem is that the following sentence does not parse: *On the receipt of the call, the station addressed should hoist the answering pennant at the dip.*

The apparent problem is the word *addressed* after *the station.* If the word *addressed* is left out, then the sentence parses normally. Examining the grammar, this word is supposed to be handled by the network $ELIDED-VERBPHR, which is called by $RELCLAUSE, which is called by $NP-SIMPLE. Since this simply does not seem to be happening, a parser TERSE-TRACE of the simple sentence *The station addressed is big* was used to find out whether the correct sequence of calls was being performed (see below). This revealed that $ELIDED-VP was indeed called on the word *addressed* and addressed was recognized as a verb past participle (VERB-PSP) by $VERB-PSP. But then it suddenly becomes clear, both from the trace and by taking a closer look at the grammar, that the passive form of the elided verb phrase network insists that there be a complement to the verb, $VCOMP-PASSIVE. This means that the sentence *The station addressed by the ship is big* should parse correctly, which it does. So the problem in this case is that the grammar was defined too restrictively.

A trial fix of this will be done simply by making the complement optional. However, it was noted in the grammar file that the complement was optional in previous versions, and the grammar had been modified to make it mandatory. There is no record for why this change was made. We will go ahead and make the change and see what happens; we will have to be on the alert for various misparses that might result from this change. So, in the grammar file, the original form of the network $ELIDED-VP-PASSIVE:

```
(NET-DEF $ELIDED-VP-PASSIVE

     (- ($VERBMOD-PASSIVE) $VERB-PSP $VCOMP-PASSIVE - ($VERBMOD-PASSIVE)
     )
)
```

will be changed to:

```
(NET-DEF $ELIDED-VP-PASSIVE
     (- ($VERBMOD-PASSIVE) $VERB-PSP - ($VCOMP-PASSIVE) - ($VERBMOD-PASSIVE)
     )
)
```

This sort of situation is perhaps one of the hardest to deal with in parsing; the grammar is not quite capturing the important constraints, and one finds oneself oscillating between different grammars, depending on which sample one is working with. It is possible that the change was previously made to make up for a lack of constraints elsewhere in the grammar, and so now the grammar can be expressed in the new, more general form. The only way to really tell will be to try to parse the same large set of sample sentences as before, to see whether something is now no longer working properly.

3.2 A Detailed Example: Quoted Noun Phrases

The problem. This example shows in detail how to add a whole new network to the grammar. Along the way, a trivial error is made that occasions tracing the parser and examining the actual ATN, before the trivial error is realized.

This problem presented itself with sentences from an NPRDC sample in which terms were enclosed in quotation marks. Thus a sentence like the following would fail to parse: *"Range" is the minimum distance from which an observer can see the light.* This sentence completely failed to parse although it is clearly a very simple sentence. The first step is to characterize the problem. Inspecting the grammar definition revealed that the only place in which quotations marks were used was the $APPSTRING network and the only place this network was referenced was in the $NP-SIMPLE network, where it is one of the options that could appear after the head noun in the slot normally occupied by a relative clause. Thus this sentence would parse: *The value "range" is the minimum distance from which an observer can see the light.* Likewise, trying the offending sentence without the quotation marks produces a correct parse. Thus it is clear that the only problem is the appearance of the quotation marks around the entire noun phrase.

Modifying the grammar. Once the problem is understood, the next step is to arrive at some specifications of how the sentence should be analyzed. Clearly, quoting a word is intended to convey something, so we should not simply ignore the quotation marks, but should convey some semantic content. Finally, the quotation marks should have minimal impact on how the parsing is done. A simple first-draft solution is to follow a general principle: try to make use of existing structure wherever possible. In this case a new network is simply defined that allows a noun phrase to be surrounded by quotation marks, and a reference to it inserted into the definition for a noun phrase; it is simply a recursive statement that one form of a noun phrase would simply be a noun phrase surrounded by quotation marks. This is a fairly general solution which would allow the different kinds of noun phrases, such as $NP-PAIR, $NP-LIST, or even $NP-STATEMENT to appear in quotation marks. Thus the following is the original grammar code:

```
(NET-DEF $NP :REGISTERS (:TESTED (?POST-VERB))
    ($NP-PAIR / $NP-SIMPLE / $NP-LIST / $NP-STATEMENT)
)
```

It is changed to the following:

```
(NET-DEF $NP :REGISTERS (:TESTED (?POST-VERB))
    ($NP-PAIR / $NP-SIMPLE / $NP-LIST / $NP-STATEMENT / !\" $NP !\")
)
```

Although this change is very simple, the basic decision is whether to define a new network, or simply elaborate the existing one. The argument in favor of defining a new network, such as one called $QUOTED-NP, is that a new network constitutes a distinct structure which the semantics module can then quite easily identify. The argument for just elaborating the existing network is basically one of avoiding proliferation of routines. One way to make the decision is to examine the semantics module code in "propositions.lisp" to determine how the eventual modifications that we'll need can be most easily made. Examining the code for $NP shows that this routine has an extremely simple structure, just simply a call for each constituent, and so it looks like a better way to implement our quoted-np would be to define a separate network. So our second draft of the grammar code is the following:

```
(NET-DEF $NP :REGISTERS (:TESTED (?POST-VERB))
    ($NP-PAIR / $NP-SIMPLE / $NP-LIST / $NP-STATEMENT / $QUOTED-NP)
)
```

Then the following new network definition is also included (this turns out to contain a syntax error):

```
(NET-DEF $QUOTED-NP :TESTED (?POST-VERB)
     (!\" $NP !\")
)
```

At this point, the tested register for the parsed verb also needs to be specified here in order to ensure that the parser takes this register setting into account. The basic reason why we know to do this is simply a matter of symmetry; all of the other noun phrase nets have this specified on it and there is no apparent reason why this one should not have it specified as well.

Testing the modification. Once the grammar is modified, the next step is to test the grammar definition by using the convenience functions in the top level of CCS. Making any such modifications is always simplified by working first with the parser until it is producing the desired parse, followed by modifying the semantics module, and then finally dealing with the criticism production rules. If any modifications need to be made to the reference module, they should also be done before the criticism rules.

A convenience function is provided in CCS for quick loads of a file named "grammar.hlg"; after starting CCS, the function (LOAD-COMPILE-HLG) is executed. This loads and compiles the current version of the grammar file. The HLG compiler prints out a list of the network names as it compiles them. Each network name appears before the compilation process is started, so if the compiler fails the problem is almost certainly somewhere in the definition of the last network name printed. The HLG compiler has only very weak error detection facilities; it catches most syntax errors, but does not provide much in the way of useful feedback. Perhaps the best suggestion for how to ensure that the syntax for the network is correct is to base new networks on modifications of existing ones.

Once the new network definition is compiled, invoke the convenience function (PARSE). This will prompt for a sentence and show the results of the parse directly on the display, and will not do any of the other processing in CCS. This allows for quick exploration of different parses. Start with a simple sentence that tests whether the grammar as a whole is still working, without intentionally involving the modified code. Thus in the example below, our first sentence will be very similar to our second test sentence, but without the quote marks.

When using the (PARSE) function, the preprocessor is not involved. For this reason, the input sentence must be LISP-readable, meaning that it has to be enclosed in parentheses, and punctuation needs to be separated from words and indicated with a backslash character to prevent it being interpreted as part of LISP syntax. Our test sentence yields a correct parse, in which *range* is considered the first noun phrase in a declarative statement.
`

```
Enter sentence or STOP:
(RANGE IS THE DISTANCE \. >END-OF-SENTENCE)
Parsing Transitions: 230
((DECLARATIVE-STATEMENT (NOUN RANGE)
  ((BE-FORM IS) ((DEFDET THE) (NOUN DISTANCE))))
 \.)
(PARSER::SUCCESS
 ($START
  ($STATEMENT
   ($STATEMENT-SIMPLE
    ($DECLARATIVE-STATEMENT ($NP ($NP-SIMPLE (NOUN RANGE)))
                            ($CLSPRED
                             ($CLSPRED-SIMPLE
                              ($COPPHR ($COPSEQ (BE-FORM IS))
                                       ($COPCOMP
                                        ($COPCOMP-SIMPLE
                                         ($NP
                                          ($NP-SIMPLE (DEFDET THE)
                                                      (NOUN DISTANCE))))))))))
   \.)
  >END-OF-SENTENCE))
```

Now we'll try surrounding the word *range* with quotation marks. Only as before, the quotation marks need to be separated out for the parse. It is clear that we are in trouble right from the beginning, because the parser takes a long time to return the result that there is a grammar problem:

```
Enter sentence or STOP:
(\" range \" is the distance \.)

(|"| RANGE |"| IS THE DISTANCE \. >END-OF-SENTENCE)
Parsing Transitions: 1033
(GRAMMAR-PROBLEM (PUNCTUATION |"|) (NOUN RANGE) (PUNCTUATION |"|)
 ((BE-FORM IS) ((DEFDET THE) (NOUN DISTANCE))) (PUNCTUATION \.))
(PARSER::SUCCESS
 ($START
  ($GRAMMAR-PROBLEM (PUNCTUATION |"|) ($NP ($NP-SIMPLE (NOUN RANGE)))
                    (PUNCTUATION |"|)
                    ($CLSPRED
                     ($CLSPRED-SIMPLE
                      ($COPPHR ($COPSEQ (BE-FORM IS))
                               ($COPCOMP
                                ($COPCOMP-SIMPLE
                                 ($NP ($NP-SIMPLE (DEFDET THE) (NOUN DISTANCE)))))))
                    (PUNCTUATION \.))
  >END-OF-SENTENCE))
```

The top-level network in the output is the grammar problem network; the quote marks are tagged as punctuation by this network, and *range* is shown as a noun phrase surrounded simply by quote mark punctuation. So for some reason, a quoted NP network was not invoked properly here. Trying a couple of variations of the sentence, such as quoting *the distance* instead of *range* showed that the failure to parse the quoted NP appeared no matter where the quoted noun phrase appeared in the sentence. Trying something simpler of course is always a good idea, so the definition of $QUOTED-NP was changed from the recursive call to $NP to simply $NP-SIMPLE as shown below:

```
(NET-DEF $QUOTED-NP :TESTED (?POST-VERB)
     (!\" $NP-SIMPLE !\")
)
```

The same sentence again still fails after reloading and compiling the grammar. This is very puzzling; it certainly should work. So it is time to bring out the big guns and do some tracing.

Tracing a parse. The source file for the parser, "parser.lisp", includes a list of some tracing flag variables, which cause various traces to be printed out. One that is generally useful for finding out what is going on is *TERSE-TRACE. This should allow us to tell when and why the QUOTED-NP network is called and whether it fails. However quite a bit of output is generated, so depending on your workstation, you need to make arrangements to be able to review this output. The tracing variable name is defined as part of the PARSER package so we need to (SETQ PARSER:*TERSE-TRACE T).

When we try our failing sentence, the result is a few thousand lines of output. Below is a sample of the first few lines with some explanation:

```
Terse 0 0 " First-time call: $START
Terse 1 0 " First-time call: $SENTENCE-TOO-COMPLEX
Terse 2 0 " Failed back to #s(PARSER::NET-FRAME :NET-NAME $SENTENCE-TOO-COMPLEX :INVOKING-ARC #s(PARSER::ARC-FRAME
:STATE $START :LAST-ARC 0 :POSITION 0) :STARTING-POSITION 0 :CHART-ENTRY-USED NIL :SCOPED-REGISTER-VALUES (?PERSON-
NUMBER INITIAL-VALUE))
Terse 2 0 " At No-more-paths from $SENTENCE-TOO-COMPLEX
Terse 2 0 " Call completely fails: $SENTENCE-TOO-COMPLEX
Terse 2 0 " First-time call: $HEADING
```

The word *Terse* is simply an identifier of which trace output this is. The first number is the contents of *TRANSITION-COUNTER, which counts how many transitions (arc crossings) through the ATN have been done. This counter is used to time out the parser and is a crude metric of how much parsing effort is required for a sentence. The second number is the current position pointer in the sentence, in which the first word is numbered 0. Following this number is the current constituent, normally a word, only here it is a quote mark. The next item in the line is the string *first-time call* which means that this is the first time the call to the following network $START is made. In the next line we see that there is a first-time call to the $SENTENCE-TOO-COMPLEX network, which is what the grammar definition specifies as the first thing the parser should check. The next line of trace output shows that we failed back to the calling frame on the stack for $SENTENCE-TOO-COMPLEX, in other words we were not able to parse using this network. The next line means that there are no more alternative ways of trying to parse that network, and the next line shows that the call to it completely fails. The next line shows that we have a first-time call to the $HEADING network, which is the next item specified by the grammar.

What we are interested in, of course, is the attempt to call the noun phrase network in the context of a declarative statement. This is in the vicinity of transition number 12, where we have a first-time call to $DECLARATIVE-STATEMENT followed by a first-time call to $NP. Then we see that the parser begins to try every possible pathway out of every possible network. At transition 55 we find the following information:

```
Terse 55 0 " First-time call: $QUOTED-NP
Terse 56 0 " Failed back to #s(PARSER::NET-FRAME :NET-NAME $QUOTED-NP :INVOKING-ARC #s(PARSER::ARC-FRAME :STATE $NP
:LAST-ARC 4 :POSITION 0) :STARTING-POSITION 0 :CHART-ENTRY-USED NIL :SCOPED-REGISTER-VALUES NIL)
Terse 56 0 " At No-more-paths from $QUOTED-NP
Terse 56 0 " Call completely fails: $QUOTED-NP
Terse 56 0 " Failed back to #s(PARSER::NET-FRAME :NET-NAME $NP :INVOKING-ARC #s(PARSER::ARC-FRAME :STATE
$DECLARATIVE-STATEMENT52 :LAST-ARC 0 :POSITION 0) :STARTING-POSITION 0 :CHART-ENTRY-USED NIL :SCOPED-REGISTER-VALUES
NIL)
Terse 56 0 " At No-more-paths from $NP
Terse 56 0 " Call completely fails: $NP
Terse 56 0 " Failed back to #s(PARSER::NET-FRAME :NET-NAME $DECLARATIVE-STATEMENT :INVOKING-ARC #s(PARSER::ARC-FRAME
:STATE $STATEMENT-SIMPLE :LAST-ARC 1 :POSITION 0) :STARTING-POSITION 0 :CHART-ENTRY-USED NIL :SCOPED-REGISTER-VALUES
(?PERSON-NUMBER INITIAL-VALUE))
Terse 56 0 " At No-more-paths from $DECLARATIVE-STATEMENT
Terse 56 0 " Call completely fails: $DECLARATIVE-STATEMENT
```

This is pretty disturbing; it shows very clearly that we called the $QUOTED-NP network and we failed to make any progress with it. This is not supposed to happen. So at this point, we will get very detailed, and along the way, illustrate how one actually looks at the compiled network.

Examining the ATN. The network can be examined simply by looking at the values of the symbols for the nodes in the ATN. Or to put it differently, the ATN is represented by values assigned to the node name symbols. The node name for the beginning node of each network is simply the name of the network, so evaluating $QUOTED-NP gives us this value:

```
> $QUOTED-NP

(#s(PARSER::ARC :TYPE PARSER::LEX :VALUE :TESTED :NEXT $QUOTED-NP163))
```

The value of a node symbol is a structure, and the various keyword items in the structure are the values assigned to the various slots in the structure. The first item, here PARSER::ARC, is simply the type of structure; namely a structure specifying an arc. The type of the arc comes next; it is a LEX arc, which means that it tests for a lexical item. The puzzling thing here is that the value slot is empty, meaning that no value was ever assigned, which is abnormal. In contrast, look at the value assigned to $NP-PAIR:

```
> $NP-PAIR

(#s(PARSER::ARC :TYPE PARSER::WORD :VALUE BOTH :NEXT $NP-PAIR136)
 #s(PARSER::ARC :TYPE PARSER::WORD :VALUE EITHER :NEXT $NP-PAIR142)
 #s(PARSER::ARC :TYPE PARSER::WORD :VALUE NEITHER :NEXT $NP-PAIR147)
 #s(PARSER::ARC :TYPE PARSER::WORD :VALUE WHETHER :NEXT $NP-PAIR152))
```

We see that this consists of four arc structures, where the type is WORD, where we are testing for a specific word in this case *both, either, neither,* or *whether.* Thus it is clear that something went wrong in the parser compiler; our first arc should be testing for a quote mark. The network $APPSTRING apparently works correctly and tests for quote marks. Taking a look at $APPSTRING seems to be a good idea:

```
> $APPSTRING

(#s(PARSER::ARC :TYPE PARSER::WORD :VALUE |"| :NEXT $APPSTRING221))
```

This is exactly what the value of $QUOTED-NP is supposed to be. Let's find out what the HLG compiler is doing.

Checking the compiler. The HLG compiler contains the key function EXPRESSION, which does the work of compiling each HLG expression. Examining the code, we see that if the first character of the first term in the expression is a "!," EXPRESSION it is supposed to call another function WORD-TEST. So the brute force approach, which the Apollo platform supports reasonably well, is simply to LISP-trace the functions PARSER::EXPRESSION and PARSER::WORD-TEST and simply wait for our $QUOTED-NP networks to come by. Comparing $APPSTRING and $QUOTED-NP will allow us to see whether the compiler is doing what it should. We need not wait for the entire compilation to complete, but can simply stop it after those networks have been traced.

Oops! As usual, brute force debugging reveals that a simple error was the problem. My error turns out embarrassingly to be nothing more than a syntax error in specifying the register in the $QUOTED-NP definition. The definition for $QUOTED-NP should have included the keyword :REGISTERS, and so should have read as follows:

```
(NET-DEF $QUOTED-NP :REGISTERS (:TESTED (?POST-VERB))
     (!\" $NP-SIMPLE !\")
```

After cleaning up and recompiling the grammar definition, we are ready to get back to where we were supposed to have been, and find out if the sentence "*Range" is the distance* is parsed correctly. It does! The correct parsing output is shown below:

```
 (\" range \" is the distance \.)

(|"| RANGE |"| IS THE DISTANCE \. >END-OF-SENTENCE)
Parsing Transitions: 289
((DECLARATIVE-STATEMENT (|"| (NOUN RANGE) |"|)
  ((BE-FORM IS) ((DEFDET THE) (NOUN DISTANCE))))
 \.)
(PARSER::SUCCESS
 ($START
  ($STATEMENT
   ($STATEMENT-SIMPLE
    ($DECLARATIVE-STATEMENT
     ($NP ($QUOTED-NP |"| ($NP-SIMPLE (NOUN RANGE)) |"|))
     ($CLSPRED
      ($CLSPRED-SIMPLE
       ($COPPHR ($COPSEQ (BE-FORM IS))
                ($COPCOMP
                 ($COPCOMP-SIMPLE
                  ($NP ($NP-SIMPLE (DEFDET THE) (NOUN DISTANCE)))))))))))
   \.)
  >END-OF-SENTENCE))
```

This parses correctly, but it uses our debugging form of $QUOTED-NP in which only $NP-SIMPLE was allowed; let's find out if our original more general definition will work:

```
(NET-DEF $QUOTED-NP :REGISTERS (:TESTED (?POST-VERB))
     (!\" $NP !\")
)
```

This appears to work; the initial noun phrase in the sentence is parsed as $NP containing $QUOTED-NP containing $NP which contains $NP-SIMPLE.

Checking the lexicon. Let's try something a little more elaborate: "*Maximum visible range" is the distance.* This fails to parse, producing the following output:

```
Enter sentence or STOP:
(\" MAXIMUM VISIBLE RANGE \" IS THE DISTANCE \.)

(|"| MAXIMUM VISIBLE RANGE |"| IS THE DISTANCE \. >END-OF-SENTENCE)
Parsing Transitions: 1323
(GRAMMAR-PROBLEM (PUNCTUATION |"|) (NOUN MAXIMUM) ((ADJ VISIBLE) (NOUN RANGE))
 (PUNCTUATION |"|) ((BE-FORM IS) ((DEFDET THE) (NOUN DISTANCE)))
 (PUNCTUATION \.))
(PARSER::SUCCESS
 ($START
  ($GRAMMAR-PROBLEM (PUNCTUATION |"|) ($NP ($NP-SIMPLE (NOUN MAXIMUM)))
                    ($NP
                     ($NP-SIMPLE ($ADJ ($ADJ-SIMPLE (ADJ VISIBLE)))
                                 (NOUN RANGE)))
                    (PUNCTUATION |"|)
                    ($CLSPRED
                     ($CLSPRED-SIMPLE
                      ($COPPHR ($COPSEQ (BE-FORM IS))
                               ($COPCOMP
                                ($COPCOMP-SIMPLE
                                 ($NP ($NP-SIMPLE (DEFDET THE) (NOUN DISTANCE))))))))
                    (PUNCTUATION \.))
  >END-OF-SENTENCE))
```

Examining this output uncovers the fact that *maximum* parsed as a noun, whereas *visible* was recognized as an adjective. Any time a puzzling parse result like this appears, one should always check the lexicon contents to make sure that words have the proper parts of speech assigned to them. This is easily done at run time just by using SYMBOL-PLIST to look at the properties of the word:

```
> (SYMBOL-PLIST 'MAXIMUM)

(WORD T ROOT T NOUN (MAXIMUM))
```

We see that the word *maximum* is only defined as a noun, so maybe that is why our last test failed to parse; we can tell from the definition of the $NP-SIMPLE that a noun followed by an adjective followed by a noun is not recognized as a legitimate noun phrase. This could be a general problem, except that we know that *maximum* should be listed as an adjective. We make a note to update the lexicon, and try our test sentence again with a different word instead of *maximum*. In this case we get a correct parse:

```
Enter sentence or STOP:
(\" USABLE VISIBLE RANGE \" IS THE DISTANCE \.)

(|"| USABLE VISIBLE RANGE |"| IS THE DISTANCE \. >END-OF-SENTENCE)
```

```
Parsing Transitions: 317
((DECLARATIVE-STATEMENT (|"| ((ADJ USABLE) (ADJ VISIBLE) (NOUN RANGE)) |"|)
  ((BE-FORM IS) ((DEFDET THE) (NOUN DISTANCE))))
 \.)
(PARSER::SUCCESS
 ($START
  ($STATEMENT
   ($STATEMENT-SIMPLE
    ($DECLARATIVE-STATEMENT
     ($NP
      ($QUOTED-NP |"|
       ($NP
        ($NP-SIMPLE ($ADJ ($ADJ-SIMPLE (ADJ USABLE)))
                    ($ADJ ($ADJ-SIMPLE (ADJ VISIBLE))) (NOUN RANGE)))
       |"|))
     ($CLSPRED
      ($CLSPRED-SIMPLE
       ($COPPHR ($COPSEQ (BE-FORM IS))
                ($COPCOMP
                 ($COPCOMP-SIMPLE
                  ($NP ($NP-SIMPLE (DEFDET THE) (NOUN DISTANCE)))))))))
   \.)
  >END-OF-SENTENCE))
```

Modifying the semantics. It looks like our parsing network is now succeeding. So now it is time to move on to the semantics module, the file "propositions.lisp", to make the appropriate modifications there. Our first modification will be very simple, to simply add a call to $QUOTED-NP as one of the constituents that $NP will look for. So the following original code will be modified:

```
(DEFTRAN $NP NIL (REFS PVPNPPHRLINKS) (REFS PVPNPPHRLINKS)
;HLG:
;    ($NP-SIMPLE / $NP-LIST / $NP-PAIR / $NP-STATEMENT)
;REFS, a list of created referents, is returned
(SCANFOR
(CONSTITUENT $NP-LIST (REFS))
(CONSTITUENT $NP-PAIR (REFS))
(CONSTITUENT $NP-STATEMENT (REFS))
(CONSTITUENT $NP-SIMPLE (REFS PVPNPPHRLINKS))
)
```

The syntax of the DEFTRAN macro needs some explanation. The macro is defined at the head of the "propositions.lisp" file and is extensively commented. Here only a brief summary will be given. Basically, functions in the semantics module translate the parse tree output of the parser into pieces of simple list structure based on Anderson's (1976) ACT representation. The parse tree is simply an embedded list of lists, where each list is headed by the name of the network that was invoked at that point in the parse. Each network has a corresponding function which analyzes the piece of the parse tree corresponding to the network. These functions simply take the contents of the remainder of the list and analyze the subparts. Eventually, a function is called that analyzes the basic word-level constituents, and assembles a set of ACT links to represent the propositional, content of that portion of the sentence. The macros DEFTRAN, SCANFOR, CONSTITUENT, and TERMINAL were defined to make these functions very easy to write by exploiting their highly repetitive and stereotyped nature.

The original $NP function shown above is defined as taking no input arguments, having local variables REFS and PVPNPPHRLINKS, and returns those two variables (using the multiple-value facility in COMMON LISP). Corresponding to the grammar definition, a noun phrase consists of one of the five listed constituents. The SCANFOR macro simply iterates through the contents of the $NP tree, and depending on which constituent is encountered, the function is called to analyzed the sub-network. Each function gets only the piece of the tree that it needs to work on. Thus, for example, $NP-SIMPLE is called, and the values of REFS and PVPNPPHRLINKS are to be returned. Our new $QUOTED-NP network should simply return the same values. So our new version of $NP is as follows:

```
(DEFTRAN $NP NIL (REFS PVPNPPHRLINKS) (REFS PVPNPPHRLINKS)
;HLG:
;    ($NP-PAIR / $NP-SIMPLE / $NP-LIST / $NP-STATEMENT / $QUOTED-NP)
;REFS, a list of created referents, is returned
(SCANFOR
(CONSTITUENT $NP-LIST (REFS))
(CONSTITUENT $NP-PAIR (REFS))
(CONSTITUENT $NP-STATEMENT (REFS))
(CONSTITUENT $NP-SIMPLE (REFS PVPNPPHRLINKS))
(CONSTITUENT $QUOTED-NP (REFS PVPNPPHRLINKS))
)
)
```

A general fact about these functions to be noted is that any constituents in the tree that are not actually scanned for and analyzed are simply ignored. Thus our definition of $QUOTED-NP is potentially extremely simple. An easy way to get the skeleton for a new function is to simply copy an existing one. The new function is as follows:

```
(DEFTRAN $QUOTED-NP NIL (REFS PVPNPPHRLINKS) (REFS PVPNPPHRLINKS)
;HLG:
;     (!\" $NP !\")
(SCANFOR
(CONSTITUENT $NP (REFS PVPNPPHRLINKS)
  (DOLIST (REF REFS) (NEW-LINK `(TAG ,REF QUOTED-NP))))
)
)
```

All this function does is simply call the original $NP network and returns everything from that network would do. However, presumably a noun phrase is quoted for some good reason which our criticism rules or integration rules may want to take into account. For this reason, the referents created by the noun phrase network will be tagged as a quoted noun phrase. Thus, the body of the CONSTITUENT macro call includes a DOLIST where each referent returned by $NP has attached to it the tag that it was part of a quoted noun phrase. Then if we choose, this tag could be made use of by the criticism rules.

Testing our new version of the proposition functions can be done very easily by simply reading into a running CCS image the new definitions of the $NP and $QUOTED-NP functions. On the Apollo this is done simply by doing a copy-paste to the LISP input window running CCS. Alternatively, one could simply load "propositions.lisp" to redefine all of the functions in the semantics module.

Once the new functions are loaded, it is then time to test to see if the correct set of propositions is defined. The top-level function (PROPOSITIONS) is like (PARSE), except the next step in the processing is done. The parse tree is handed to the semantics module and the returned propositions are displayed. This output is shown below:

```
> (PROPOSITIONS)
. . .

Enter sentence or STOP:
(\" RANGE \" IS THE DISTANCE \.)

(|"| RANGE |"| IS THE DISTANCE \. >END-OF-SENTENCE)
Parsing Transitions: 293
((DECLARATIVE-STATEMENT (|"| (NOUN RANGE) |"|)
  ((BE-FORM IS) ((DEFDET THE) (NOUN DISTANCE))))
 \.)
(PARSER::SUCCESS
 ($START
  ($STATEMENT
   ($STATEMENT-SIMPLE
    ($DECLARATIVE-STATEMENT
     ($NP ($QUOTED-NP |"| ($NP ($NP-SIMPLE (NOUN RANGE))) |"|))
     ($CLSPRED
      ($CLSPRED-SIMPLE
       ($COPPHR ($COPSEQ (BE-FORM IS))
                ($COPCOMP
                 ($COPCOMP-SIMPLE
                  ($NP ($NP-SIMPLE (DEFDET THE) (NOUN DISTANCE)))))))))
   \.)
  >END-OF-SENTENCE))
Pronoun antecedents:
NIL
ACT Propositions:
((S PROP1 REF1) (P PROP1 RANGE) (S PROP3 REF1) (P PROP3 REF2) (S PROP2 REF2)
 (P PROP2 DISTANCE))

ALL Propositions:
((TAG PROP3 SENTENCE-MAIN) (TAG REF1 QUOTED-NP) (TAG REF1 COMPLETELY-NEW)
 (TAG REF1 NEW-REFERENT) (TAG REF1 NP-REFERENT) (TAG PROP1 NP-PROP)
 (S PROP1 REF1) (P PROP1 RANGE) (TAG PROP1 HEAD-NOUN) (TAG REF1 NODETERMINER)
 (NUMBER REF1 SINGULAR) (GENDER REF1 N) (SRF REF1 (RANGE)) (HNF REF1 RANGE)
 (NP-PROP-NUMBER REF1 1) (TAG REF1 STATEMENT-SUBJECT) (S PROP3 REF1)
 (P PROP3 REF2) (TAG REF2 COMPLETELY-NEW) (TAG REF2 NEW-REFERENT)
 (TAG REF2 NP-REFERENT) (TAG PROP2 NP-PROP) (S PROP2 REF2) (P PROP2 DISTANCE)
 (TAG PROP2 HEAD-NOUN) (TAG REF2 DEFINITE) (NUMBER REF2 SINGULAR)
 (GENDER REF2 N) (SRF REF2 (DISTANCE)) (HNF REF2 DISTANCE)
 (NP-PROP-NUMBER REF2 1) (TAG PROP3 STATEMENT-MAIN)
 (TAG SENTENCE-PROP-NUMBER 3))
```

It is often helpful to draw a semantic network diagram of the proposition links. The ACT propositions contain just the basic ACT links. Here we see that referent 1 (REF1) has predicated of it with PROP1 that it is *range,* and also with PROP3 that it is REF2, which in turn has predicated (PROP3) of it that it is *distance.* So this is saying that range has the property of being a distance, which is a correct semantic structure.

The list of all propositions shows the various tags that are attached to these basic links. Many of these tags are used by the criticism rules and also play a role in reference resolution. Here what we are concerned with is whether our new tag of QUOTED-NP was attached to the correct place. Sure enough, the item (TAG REF1 QUOTED-NP) appears in the list. So we have successfully modified the propositions file.

Tracing semantics functions. If things had not gone well, a good approach is to trace the individual functions in the semantics module, such as $NP and $NP-SIMPLE, as well as our new function $QUOTED-NP, to examine the tree structure that is handed to each function and whether the anticipated results were returned. This output for the same sentence is shown below:

```
(\" RANGE \" IS THE DISTANCE \.)

(|"| RANGE |"| IS THE DISTANCE \. >END-OF-SENTENCE)
Parsing Transitions: 293
((DECLARATIVE-STATEMENT (|"| (NOUN RANGE) |"|)
  ((BE-FORM IS) ((DEFDET THE) (NOUN DISTANCE))))
 \.)
(PARSER::SUCCESS
 ($START
  ($STATEMENT
   ($STATEMENT-SIMPLE
    ($DECLARATIVE-STATEMENT
     ($NP ($QUOTED-NP |"| ($NP ($NP-SIMPLE (NOUN RANGE))) |"|))
     ($CLSPRED
      ($CLSPRED-SIMPLE
       ($COPPHR ($COPSEQ (BE-FORM IS))
                ($COPCOMP
                 ($COPCOMP-SIMPLE
                  ($NP ($NP-SIMPLE (DEFDET THE) (NOUN DISTANCE))))))))))
   \.)
  >END-OF-SENTENCE))
Pronoun antecedents:
(NON-SUBJECT (SINGULAR (N (REF2))) SUBJECT (SINGULAR (N (REF1))))
 0: ($NP (($QUOTED-NP |"| ($NP ($NP-SIMPLE (NOUN RANGE))) |"|)))
   1: ($QUOTED-NP (|"| ($NP ($NP-SIMPLE (NOUN RANGE))) |"|))
     2: ($NP (($NP-SIMPLE (NOUN RANGE))))
       3: ($NP-SIMPLE ((NOUN RANGE)))
       3: returned ((TAG REF3 COMPLETELY-NEW) (TAG REF3 NEW-REFERENT)
          (TAG REF3 NP-REFERENT) (TAG PROP4 NP-PROP) (S PROP4 REF3)
          (P PROP4 RANGE) (TAG PROP4 HEAD-NOUN) (TAG REF3 NODETERMINER)
          (NUMBER REF3 SINGULAR) (GENDER REF3 N) (SRF REF3 (RANGE))
          (HNF REF3 RANGE) (NP-PROP-NUMBER REF3 1)) (REF3) NIL
     2: returned ((TAG REF3 COMPLETELY-NEW) (TAG REF3 NEW-REFERENT)
                  (TAG REF3 NP-REFERENT) (TAG PROP4 NP-PROP) (S PROP4 REF3)
                  (P PROP4 RANGE) (TAG PROP4 HEAD-NOUN) (TAG REF3 NODETERMINER)
                  (NUMBER REF3 SINGULAR) (GENDER REF3 N) (SRF REF3 (RANGE))
                  (HNF REF3 RANGE) (NP-PROP-NUMBER REF3 1)) (REF3) NIL
   1: returned ((TAG REF3 QUOTED-NP) (TAG REF3 COMPLETELY-NEW) (TAG REF3 NEW-REFERENT)
                (TAG REF3 NP-REFERENT) (TAG PROP4 NP-PROP) (S PROP4 REF3)
                (P PROP4 RANGE) (TAG PROP4 HEAD-NOUN) (TAG REF3 NODETERMINER)
                (NUMBER REF3 SINGULAR) (GENDER REF3 N) (SRF REF3 (RANGE))
                (HNF REF3 RANGE) (NP-PROP-NUMBER REF3 1)) (REF3) NIL
 0: returned ((TAG REF3 QUOTED-NP) (TAG REF3 COMPLETELY-NEW) (TAG REF3 NEW-REFERENT)
               (TAG REF3 NP-REFERENT) (TAG PROP4 NP-PROP) (S PROP4 REF3)
               (P PROP4 RANGE) (TAG PROP4 HEAD-NOUN) (TAG REF3 NODETERMINER)
               (NUMBER REF3 SINGULAR) (GENDER REF3 N) (SRF REF3 (RANGE))
               (HNF REF3 RANGE) (NP-PROP-NUMBER REF3 1)) (REF3) NIL
 0: ($NP (($NP-SIMPLE (DEFDET THE) (NOUN DISTANCE))))
 ...
ACT Propositions:
((S PROP4 REF3) (P PROP4 RANGE) (S PROP6 REF3) (P PROP6 REF4) (S PROP5 REF4)
 (P PROP5 DISTANCE))

...
```

For brevity, the trace was abbreviated for the second noun phrase, and the ALL propositions output was deleted. But here you can see in each level of the trace that the first call to $NP was given a piece of tree beginning with $QUOTED-NP. The $QUOTED-NP function was then given a piece of tree starting with a quote mark and including $NP. Our $QUOTED-NP function ignored the quote mark (which is redundant with the fact that we are in the $QUOTED-NP function), and the $NP function was then handed the tree beginning $NP-SIMPLE, which function was in turn handed the tree that just included (NOUN RANGE). $NP-SIMPLE then returned the bulk of the propositions and $QUOTED-NP simply added the QUOTED-NP tag.

Conclusion. The revision that we have at this point thus allows a quoted noun phrase to appear anywhere a noun phrase would appear, builds the same structure it would if the quotes had not appeared, except the referent of the noun phrase is tagged as having appeared as a quoted noun phrase. This would suffice to simply allow sentences of quoted noun phrases to be analyzed as if the quotes had not appeared. However if we wish to pay attention in the criticism rules to the quoted noun phrases we need to have some notion of what comment or criticism we should make.

Subsequent trials showed that this set of definitions was defective, because the quote marks are required to enclose the *entire* noun phrase. So *The "range" is the distance* would fail. So these definitions were discarded in favor of simply allowing quote marks to appear inside $NP-SIMPLE, whose semantics function then adds the tag if it finds a quote mark.

3.3 A Summary Example: Imperative Copulative Sentences

The following sentence failed to parse - there was no provision for imperative sentences based on the verb *to be*: *Be aware that this situation could arise.*

The original imperative statement network called a network for handling the special form of an imperative statement verb and its complement:

```
(NET-DEF $IMPERATIVE-STATEMENT :REGISTERS (:SCOPED (?PERSON-NUMBER ?POST-VERB))
     (- ($INITIAL-VERBMOD - (!\,)) $IMPERATIVE-VERBPHR)
)
```

We could handle our problem by just calling $COPPHR, which is the network that analyzes the general copulative predicate phrase. The problem is that it accepts a variety of constructions that would be ungrammatical in this form, such as *Are aware that this situation could arise.* The corresponding problem is why imperative sentences were defined with the specialized imperative verb phrase network. We will try the same pattern here, where the copulative phrase for imperative sentences will require an infinitive form of the verb *to be* and will otherwise be similar in structure to $COPPHR.

Here is the new form of $IMPERATIVE-STATEMENT and the new network $IMPERATIVE-COPPHR:

```
(NET-DEF $IMPERATIVE-STATEMENT :REGISTERS (:SCOPED (?PERSON-NUMBER ?POST-VERB))
     (- ($INITIAL-VERBMOD - (!\,))
        ($IMPERATIVE-VERBPHR / $IMPERATIVE-COPPHR))
)

(NET-DEF $IMPERATIVE-COPPHR
 (BE-INF - ($VERBMOD) $COPCOMP - ($VERBMOD))
)
```

The new nets appeared to work correctly, so the next step is to modify the semantics code. Below is the original form of the $IMPERATIVE-STATEMENT function:

```
(DEFTRAN $IMPERATIVE-STATEMENT NIL
 (REF PROPS RELS MAIN-PROPS REL-NODES)
 (MAIN-PROPS)
;output is the main proposition node(s) from the clause predicates
(SETQ REF 'SELF)
(NEW-LINK `(TAG ,REF STATEMENT-SUBJECT))
(SCANFOR
(CONSTITUENT ($IMPERATIVE-VERBPHR STS.GENERAL-VERBPHR)
   ((PROPS REL-NODES)(NIL REF 'UNSPECIFIED))
(SETQ MAIN-PROPS (APPEND MAIN-PROPS PROPS))
(SETQ REL-NODES (APPEND REL-NODES RELS)))
)
;If there were initial verb modifiers, apply them to each clause predicate
;main proposition and rel-node.  We don't care about the propositions for the verb modifiers
;themselves
(SCANFOR
(CONSTITUENT ($INITIAL-VERBMOD $VERBMOD) (NIL (MAIN-PROPS REL-NODES)))
)
;Maybe only one of MAIN-PROPS should get returned as the main proposition
)
```

This function illustrates a useful technique and principle. Notice how the constituent $IMPERATIVE-VERBPHR is actually processed by the function STS.GENERAL-VERBPHR. Once the grammar has properly parsed a particular string, the constraints that the grammar imposes are no longer relevant to how the semantic structure should be built for the parse tree. Thus, STS.GENERAL-VERBPHR is a function that builds semantic structure for a very broad class of verb phrases, being quite unconcerned with what syntactic constraints should be present. Thus the constraints of grammar need only be reflected in the grammar definition, not in the propositions functions. The new imperative statement function takes advantage of the same principle to analyze the output of the new $IMPERATIVE-COPPHR network just by using the existing the existing $COPPHR function:

```
(DEFTRAN $IMPERATIVE-STATEMENT NIL
 (REF PROPS RELS MAIN-PROPS REL-NODES)
 (MAIN-PROPS)
;output is the main proposition node(s) from the clause predicates
(SETQ REF 'SELF)
(NEW-LINK `(TAG ,REF STATEMENT-SUBJECT))
(SCANFOR
(CONSTITUENT ($IMPERATIVE-VERBPHR STS.GENERAL-VERBPHR)
   ((PROPS REL-NODES)(NIL REF 'UNSPECIFIED))
 (SETQ MAIN-PROPS (APPEND MAIN-PROPS PROPS))
 (SETQ REL-NODES (APPEND REL-NODES RELS)))
(CONSTITUENT ($IMPERATIVE-COPPHR $COPPHR)((PROPS)(REF))
 (SETQ MAIN-PROPS (APPEND MAIN-PROPS PROPS))
;make REL-NODES the same as props to show that all VERBMODS will be attached to
;propositions
 (SETQ REL-NODES (APPEND REL-NODES PROPS)))
)
;If there were initial verb modifiers, apply them to each clause predicate
;main proposition and rel-node.  We don't care about the propositions for the verb modifiers
;themselves
(SCANFOR
(CONSTITUENT ($INITIAL-VERBMOD $VERBMOD) (NIL (MAIN-PROPS REL-NODES)))
)
;TAG that these were expressed imperatively
(DOLIST (PROP MAIN-PROPS)(NEW-LINK `(TAG ,PROP IMPERATIVE)))
;Maybe only one of MAIN-PROPS should get returned as the main proposition
)
```

This set of modifications appears to work correctly but the parse and propositions are still not correct for the original sentence *Be aware that this situation could arise.*   A basic reason why is that for some inexplicable reason *aware* is shown in the lexicon only as a noun. But the sentence *Be happy that this situation could arise* also fails to parse, and we now see that the copulative sentence form will not allow the sentences of the form subject <*subject*> *is* <*adjective*> *that* <*statement*>. This is another substantial hole in the grammar, but one that did not previously appear. Incorporating this requires first, a modification to the grammar to allow this type of sentence to appear, and second, a further addition to the propositions function for copulative complements that builds a *cause* relationship between the embedded statement and the outer proposition. Such sentences are actually a form of complex sentence, being paraphrasable as *Because this situation could arise, be aware.* Thus a better place to make this set of changes might be to form a new kind of complex sentence; however the new definitions would then be nearly duplicated between this type of complex sentence and this imperative form of copulative sentence. So instead, the changes were made to $COPCOMP-SIMPLE and its corresponding propositions functions. These changes are not shown in this example.

3.4. Adding a Simple Criticism Rule

The new criticism. This example is a simple very one, and is perhaps not very realistic, but it will serve to introduce the basic ideas about how one goes about adding a new criticism rule to the system.

Consider this sentence from an actual piece of Navy material: *The proper installation and maintenance of the various electrical systems aboard ship are very important to the electrician's mate.* The basic form of this sentence is *Cables are important to the electrician's mate.* Such sentences can be criticized being "weak" sentences. The author probably had something more pertinent in mind but could not express it very well. The object of this example will be to install a criticism rule that will detect and comment on sentences like this. We will base the rule on a fairly specific form, namely, the use of the adjective *important* in a sentence based on the verb *to be;* that is, sentences of the form *<subject> <is> important <other material>*. Of course some simpler text-criticism programs could detect such patterns, but this system could detect this basic structure even if the word *important* is surrounded by a variety of other material.

Checking available information. The first step is to determine whether the parser and the semantics module already supply the necessary information. The way to do this is to use the interactive mode of CCS, and try various sample sentences and examine the parse output and the propositions output. For example, the sentence *The job is important* produces the following output:

```
> (propositions)
. . .
Enter sentence or STOP:
(the job is important \.)

(THE JOB IS IMPORTANT \. >END-OF-SENTENCE)
Parsing Transitions: 164
((DECLARATIVE-STATEMENT ((DEFDET THE) (NOUN JOB))
  ((BE-FORM IS) (ADJ IMPORTANT)))
 \.)
(PARSER::SUCCESS
 ($START
  ($STATEMENT
   ($STATEMENT-SIMPLE
    ($DECLARATIVE-STATEMENT ($NP ($NP-SIMPLE (DEFDET THE) (NOUN JOB)))
                            ($CLSPRED
                             ($CLSPRED-SIMPLE
                              ($COPPHR ($COPSEQ (BE-FORM IS))
                                       ($COPCOMP
                                        ($COPCOMP-SIMPLE
                                         ($ADJ ($ADJ-SIMPLE (ADJ IMPORTANT)))))))))
   \.)
  >END-OF-SENTENCE))
Pronoun antecedents:
NIL
ACT Propositions:
((S PROP1 REF1) (P PROP1 JOB) (S PROP2 REF1) (P PROP2 IMPORTANT))

ALL Propositions:
((TAG PROP2 SENTENCE-MAIN) (TAG REF1 COMPLETELY-NEW) (TAG REF1 NEW-REFERENT)
 (TAG REF1 NP-REFERENT) (TAG PROP1 NP-PROP) (S PROP1 REF1) (P PROP1 JOB)
 (TAG PROP1 HEAD-NOUN) (TAG REF1 DEFINITE) (NUMBER REF1 SINGULAR)
 (GENDER REF1 N) (SRF REF1 (JOB)) (HNF REF1 JOB) (NP-PROP-NUMBER REF1 1)
 (TAG REF1 STATEMENT-SUBJECT) (S PROP2 REF1) (P PROP2 IMPORTANT)
 (TAG PROP2 STATEMENT-MAIN) (TAG SENTENCE-PROP-NUMBER 2))
```

Here we see that the basic structure of a sentence that simply modifies something as being important is an S link to a referent and a P link to the predicate IMPORTANT. But note that a sentence like *The important job is difficult* would also show something as being modified by IMPORTANT. To distinguish these two cases, the SENTENCE-MAIN tag can be used. In the above example, we see that this tag is attached to PROP2, which is the proposition node for the IMPORTANT predicate. There are other potentially useful tags as well. The STATEMENT-MAIN and STATEMENT-SUBJECT tags identify the main proposition and subject of all statements embedded or top-level. However, the SENTENCE-MAIN tag is the one we want to use here because it identifies the top-level statement.

Adding the new rule. The second step is to draft a production rule for the criticism that matches the pattern that we have chosen from the propositions and insert this into the set of production rules.

Examining the production rules file "compsys.prs" shows a first section titled "Grammatical and sentence-level criticisms". The criticism that we have in mind is basically a criticism at the level of the single sentence, as opposed to criticisms about the referential content of the sentence, or how it relates to the topic structure. Examining further, we see a rule relatively close to the one we would like, namely the rule NEGATED-MAIN CLAUSE, which criticizes a sentence whose main clause is negated, such as *The job is not important.* This rule is shown below:

```
(NEGATED-MAIN-CLAUSE
IF ((STEP CRITICIZE SENTENCE-LEVEL)
     (SM TAG ?P1 SENTENCE-MAIN)
     (SM S ?P2 ?P1)(SM P ?P2 FALSE) )
THEN ((WRITE-MESSAGE NEGATED-MAIN-CLAUSE CRITICISM
        "The main clause of this sentence is negative, which can be hard to understand.~
         ~%Can you rewrite the sentence into a positive form?"
        "Try to rewrite into positive form.")
))
```

An explanation of this rule will help in constructing the new rule. The first term in the expression is simply the name of the rule; then there is the symbol IF, followed by an expression for the production rule condition, and then the symbol THEN followed by the production rule action. The components of the condition, the *clauses*, simply match a pattern in the *Sentence Memory* (indicated by the term SM), which contains the proposition links of the sentence. The first clause is a piece of control information; other production rules cause clauses like this one to be put in to or taken out of the production system database in order to control the order in which the rules fire.

The pattern that this rule looks for is that the SENTENCE-MAIN proposition node, which is assigned to the variable ?P1, has attached to it another proposition, ?P2, that has the predicate FALSE. If the pattern is present for some values of the variables ?P1 and ?P2, this rule fires and the message is output.

The message consists of two strings, which follow the rules for the COMMON LISP FORMAT function. The first string is the *verbose* message form, and the second is the *terse* message form. The first argument of the WRITE-MESSAGE function is a tag that identifies the message. This is used by the verbose/terse mechanism to determine whether this message has been delivered before. The second argument is either CRITICISM or INFORMATION, which is used by the output options mechanism to determine whether only criticism messages or both criticism and information messages appear.

Our new rule for weak sentences will clearly be very similar to this one; the only difference is that instead of testing for a false predication on the main proposition, we will simply be testing for whether the main proposition consisted of the predicate IMPORTANT. Here is our draft production rule:

```
(OF-COURSE-IT-IS-IMPORTANT
IF ((STEP CRITICIZE SENTENCE-LEVEL)
     (SM TAG ?P1 SENTENCE-MAIN)
     (SM P ?P1 IMPORTANT) )
THEN ((WRITE-MESSAGE WEAK-SENTENCE CRITICISM
        "The main clause of this sentence simply says that something is important,~
         ~%Can you make the sentence more to the point?"
        "Try to make more to the point.")
))
```

This title for the rule is flippant, but it is normally invisible to the user. The wording of the message tag WEAK-SENTENCE and the actual message text needs to be chosen carefully, because this is the interface between the user and CCS. If these messages do not convey the meaning of the criticism, and what the writer should do about it, then the criticism will not be useful. These are not issues of programming, so they will not be dealt with further here.

We will insert the new rule into the "compsys.prs" file right after the negated main clause rule. Generally, it is not very important where the rule appears in the production system file, because the production system interpreter does not rely on ordering of the rules. However, for ease of programming it is best to group related rules together.

Testing the new rule. After the new rule has been inserted into the file, we are ready to try it out. The only provision for installing a new rule is to compile and install the entire set of production rules. The way to do this on a permanent basis is to build a new image of CCS. However, this is very time consuming, so a better debugging strategy is to invoke CCS and temporarily recompile and install the production rule set using the convenience function (LOAD-COMPILE-PS). This gives the result shown below:

```
> (load-compile-ps)

RULE COMPILED: CPS-CRITICIZE-SENTENCE-LEVEL
RULE COMPILED: CPS-SENTENCE-NONPARSABLE
...
RULE COMPILED: CPS-MISSING-DETERMINERS
RULE COMPILED: CPS-NEGATED-MAIN-CLAUSE
RULE COMPILED: CPS-OF-COURSE-IT-IS-IMPORTANT
RULE COMPILED: CPS-MULTIPLE-NEGATION
...
RULE COMPILED: CPS-CLEANUP-FIVES
RULE COMPILED: CPS-PROCESSING-DONE
"PPS compilation done"
>
```

Something to look for is whether the new rule is listed as compiled which is the case here. If something is seriously wrong with the syntax of the rule, error messages will normally appear; if nothing happens, or the compiler hangs, one should of course suspect the new rule, and the list of which rules are compiled can tell you how far along the compiler got before the problem appeared.

We will now try out our new rule by invoking the interactive convenience function (CRITICIZE) and providing an input sentence. Debugging aids for this step show the trace of the production system as it is executing. These can be turned on by means of the global variables listed in the file "ccs-build.lisp". In particular, the variables PPS:*VERBOSE-TRACE and PPS:*TERSE-TRACE turn on and off a verbose or terse trace of the production system activity. If you do not have the multiple window facilities implemented, then you will be interested in the variable PPS:*TOP-LEVEL-ALL-OUTPUT which, if non-NIL, causes the contents of the verbose trace and terse trace to appear on the top-level LISP window. The best strategy is to try running a new rule set without any tracing, especially if the change is fairly simple, because the traces are quite lengthy. Here is the results from trying out our new rule without tracing:

```
> (criticize)
...
Enter sentence or STOP:
(the ship is important \.)

(THE SHIP IS IMPORTANT \. >END-OF-SENTENCE)
Parsing Transitions: 169
Pronoun antecedents:
NIL
ACT Propositions:
((S PROP1 REF1) (P PROP1 SHIP) (S PROP2 REF1) (P PROP2 IMPORTANT))

ALL Propositions:
((TAG PROP2 SENTENCE-MAIN) (TAG REF1 COMPLETELY-NEW) (TAG REF1 NEW-REFERENT)
 (TAG REF1 NP-REFERENT) (TAG PROP1 NP-PROP) (S PROP1 REF1) (P PROP1 SHIP)
 (TAG PROP1 HEAD-NOUN) (TAG REF1 DEFINITE) (NUMBER REF1 SINGULAR)
 (GENDER REF1 N) (SRF REF1 (SHIP)) (HNF REF1 SHIP) (NP-PROP-NUMBER REF1 1)
 (TAG REF1 STATEMENT-SUBJECT) (S PROP2 REF1) (P PROP2 IMPORTANT)
 (TAG PROP2 STATEMENT-MAIN) (TAG SENTENCE-PROP-NUMBER 2))


WEAK-SENTENCE
The main clause of this sentence simply says that something is important,
Can you make the sentence more to the point?

...
```

Success! Our new rule fired right away. If it hadn't, we would need to debug using the trace.   For this next example I deliberately crippled the rule to keep it from working. We set up the trace and start the criticism:

```
(OF-COURSE-IT-IS-IMPORTANT
IF ((STEP CRITICIZE SENTENCE-LEVEL)
     (SM TAG ?P1 MAIN-PROPOSITION)
     (SM P ?P1 IMPORTANT) )
THEN ((WRITE-MESSAGE WEAK-SENTENCE CRITICISM
        "The main clause of this sentence simply says that something is important,~
         ~%Can you make the sentence more to the point?"
        "Try to make more to the point.")
))
> (setq pps:*toplevel-all-output T)

T
> (setq pps:*verbose-trace T)

T
> (criticize)

...
```

```
Enter sentence or STOP:
(the ship is important \.)
```

A very large amount of material appears; this is a good place to use any "dribble" facilities in your LISP implementation. The output consists of a listing of the contents of the production system database on each production system cycle, along with a list of which rules were fired and what the variable bindings were for them. Here is shown just the output from the relevant cycle. We know what the relevant cycle is because we know that the rule will only trigger if the clause (STEP CRITICIZE SENTENCE-LEVEL) is present. So we simply find the cycle in which we have the corresponding step listed as database contents. In this example this turns out to be on cycle 2; the trace is shown below:

```
...
Cycle 2
GOAL
STEP
   (STEP CRITICIZE SENTENCE-LEVEL)
WM
LTM
PM
   (PM TAG UNSPECIFIED-GLOBAL-TOPIC GLOBAL-TOPIC)
   (PM TAG UNSPECIFIED-SUBTOPIC SUBTOPIC)
   (PM TAG UNSPECIFIED-PARAGRAPH-TOPIC PARAGRAPH-TOPIC)
   (PM TAG UNSPECIFIED-LOCAL-TOPIC LOCAL-TOPIC)
   (PM TOPIC-STRUCTURE UNSPECIFIED-GLOBAL-TOPIC UNSPECIFIED-SUBTOPIC)
   (PM TOPIC-STRUCTURE UNSPECIFIED-SUBTOPIC UNSPECIFIED-PARAGRAPH-TOPIC)
   (PM TOPIC-STRUCTURE UNSPECIFIED-PARAGRAPH-TOPIC UNSPECIFIED-LOCAL-TOPIC)
   (PM TAG UNSPECIFIED-SUBTOPIC CURRENT-SUBTOPIC)
   (PM TAG UNSPECIFIED-PARAGRAPH-TOPIC CURRENT-PARAGRAPH-TOPIC)
   (PM TAG UNSPECIFIED-LOCAL-TOPIC CURRENT-LOCAL-TOPIC)
SM
   (SM TAG PROP2 SENTENCE-MAIN)
   (SM TAG REF1 COMPLETELY-NEW)
   (SM TAG REF1 NEW-REFERENT)
   (SM TAG REF1 NP-REFERENT)
   (SM TAG PROP1 NP-PROP)
   (SM S PROP1 REF1)
   (SM P PROP1 SHIP)
   (SM TAG PROP1 HEAD-NOUN)
   (SM TAG REF1 DEFINITE)
   (SM NUMBER REF1 SINGULAR)
   (SM GENDER REF1 N)
   (SM SRF REF1 (SHIP))
   (SM HNF REF1 SHIP)
   (SM NP-PROP-NUMBER REF1 1)
   (SM TAG REF1 STATEMENT-SUBJECT)
   (SM S PROP2 REF1)
   (SM P PROP2 IMPORTANT)
   (SM TAG PROP2 STATEMENT-MAIN)
   (SM TAG SENTENCE-PROP-NUMBER 2)
COMMENT
Fired:
   CPS-FIND-SENTENCE-SUBJECT-1
   CPS-NOMINATE-NEW-MAIN-PROP-ANTECEDENT-2
   CPS-OUTPUT-SENTENCE-LEVEL
Rule Fired: CPS-FIND-SENTENCE-SUBJECT-1
Bindings: ((?P PROP2 ?R REF1))
Rule Actions:
ADD: (SM TAG REF1 SENTENCE-SUBJECT)
Rule Fired: CPS-NOMINATE-NEW-MAIN-PROP-ANTECEDENT-2
Bindings: ((?P PROP2 ?R REF1 ?N IMPORTANT))
Rule Actions:
Rule Fired: CPS-OUTPUT-SENTENCE-LEVEL
Rule Actions:
DELETE: (STEP CRITICIZE SENTENCE-LEVEL)
ADD: (STEP OUTPUT SENTENCE-LEVEL)
```

The first part is the different categories of information present in the production system database. By convention, the first item in a clause (the "tag") is the "type" of information. In CCS, two especially important types are PM for *passage memory* and SM for *sentence memory.* So this is the cycle on which our rule is supposed to fire. If we look down to the list of rules that are fired, we see that our new rule did not fire. To find out why, we compare each clause in the rule with the contents of PM and SM to determine if indeed the clauses were present. Most errors in programming production rules are in writing a condition that is not in fact satisfied when it is supposed to be, either because previous rules or previous processing did not insert the right clauses, or because the rule tests for the wrong clauses.

Systematically looking at the clauses in our rule we see that the STEP clause is present, but looking for the clauses starting with SM TAG, we see that there is no clause containing MAIN PROPOSITION, which this erroneous version of the rule tests for. On the other hand we see that there is definitely a clause that matches the one about a proposition being important. So our only difficulty here is that the defective rule looks for MAIN-PROPOSITION when it should be looking for SENTENCE-MAIN, which is, of course, what our original rule tested for. If we make this change then the rule should work again.

Below is the condensed trace from a correct rule:

```
Cycle 2
GOAL
STEP
   (STEP CRITICIZE SENTENCE-LEVEL)
WM
LTM
PM
...
SM
   (SM TAG PROP2 SENTENCE-MAIN)
...
   (SM TAG REF1 STATEMENT-SUBJECT)
   (SM S PROP2 REF1)
   (SM P PROP2 IMPORTANT)
   (SM TAG PROP2 STATEMENT-MAIN)
COMMENT
Fired:
   CPS-OF-COURSE-IT-IS-IMPORTANT
   CPS-FIND-SENTENCE-SUBJECT-1
   CPS-NOMINATE-NEW-MAIN-PROP-ANTECEDENT-2
   CPS-OUTPUT-SENTENCE-LEVEL
Rule Fired: CPS-OF-COURSE-IT-IS-IMPORTANT
Bindings: ((?P1 PROP2))
Rule Actions:
 The main clause of this sentence simply says that something is important,
Can you make the sentence more to the point?

WEAK-SENTENCE
The main clause of this sentence simply says that something is important,
Can you make the sentence more to the point?
Rule Fired: CPS-FIND-SENTENCE-SUBJECT-1
Bindings: ((?P PROP2 ?R REF1))
Rule Actions:
ADD: (SM TAG REF1 SENTENCE-SUBJECT)
Rule Fired: CPS-NOMINATE-NEW-MAIN-PROP-ANTECEDENT-2
Bindings: ((?P PROP2 ?R REF1 ?N IMPORTANT))
Rule Actions:
Rule Fired: CPS-OUTPUT-SENTENCE-LEVEL
Rule Actions:
DELETE: (STEP CRITICIZE SENTENCE-LEVEL)
ADD: (STEP OUTPUT SENTENCE-LEVEL)
```

By way of a little further explanation, the trace contains a list of the names of rules that have fired, and then for each fired rule, there is a list of the bindings of any variables that appeared in the rule. So we see for our new rule that the variable ?P1 was assigned to the value PROP2. This list of binding information can be useful, especially when there are multiple bindings that satisfy a rule. Following the list of bindings is the rule actions, which in this case is the returned value of the WRITE-MESSAGE function and the actual output of the WRITE-MESSAGE function appears in the top-level window. The rest of the example shows other rules that were fired and what their bindings are. Notice that one of the rules binds three different variables, and that some of the actions consist of adding and deleting step information to the database to take the production system on to the next phase.

Elaborating the new rule. Now that the rule appears to work, it is important to try it out on a variety of other sentences to gain confidence that it does what we want it to do. For example, here is a trial which shows how the CCS system can find the vacuous main proposition in spite of a relatively elaborate sentence:

```
Enter sentence or STOP:
(the job performed by the electrician is always extremely important to the effectiveness
of the ship in combat situations \.)

...

WEAK-SENTENCE
The main clause of this sentence simply says that something is important,
Can you make the sentence more to the point?

SENTENCE-TOO-BIG
The sentence contains 13 propositions of new information, which may be too
much to understand easily.  Check whether all of the information is actually
necessary at this point, and if so, try expressing it with smaller sentences.

...
```

This sentence, which looks just like a typical piece of Navy writing, triggers both the weak sentence criticism and also the sentence-too-big criticism, among others. Notice that when testing sentences in this situation, many criticisms will appear that are related to the coherence (or lack of it) of the sentence with previous material. Normally, you would ignore such messages, because you are testing single sentences out of context. If we were testing for criticisms where context is important of course, we would need to supply previous sentences.

Let's add a refinement to this rule that will also illustrate some of the other mechanisms available. Let's have the rule identify the sentence subject. Here is the revised production rule:

```
(OF-COURSE-IT-IS-IMPORTANT
IF ((STEP CRITICIZE SENTENCE-LEVEL)
     (SM TAG ?P1 SENTENCE-MAIN)
     (SM P ?P1 IMPORTANT)
     (SM S ?P1 ?R)
     (SM TAG ?R STATEMENT-SUBJECT)
)
THEN ((FRF-SM-PM ?R ^R)
      (WRITE-MESSAGE WEAK-SENTENCE CRITICISM
        "The main clause of this sentence simply says that ~A ~A is important,~
         ~%Can you make the sentence more to the point?"
        "Try to say something more to the point about ~A ~A."
        ?R ^R)
))
```

The condition of this rule is elaborated to find the referent node, ?R, at the end of the S link from the proposition node, ?P1. ?R is also tagged as STATEMENT-SUBJECT. Thus ?R will be assigned the value of the referent node corresponding to the statement subject. Then in the action of the rule, the function FRF-SM-PM looks in the production rule database for the special forms that are available for describing referents. It looks for the SRF form, which is a simple noun phrase word string that was actually used to introduce or describe the referent, or if that is not present, it looks for the HNF, which is the original head noun used to describe the referent. The purpose of this is to provide an intelligible description of the subject of the sentence. Simply showing "REF2" would not be very useful.

In this example, the function takes a variable from the production rule condition, and finds the referential form, and supplies that to the value of the local action variable ^R. Note that the PPS interpreter requires local action variables to be designated with a "^" prefix; these are variables whose bindings exist only during the execution of a production rule action. In contrast, the variables designated with a "?" have values defined only over the entire production rule. Of course, other variables can be used, but then the programmer will be completely responsible for ensuring that their values always have appropriate scope.

The message in our rule follows FORMAT function rules to print out the values of these variables with ~A items. By convention, whenever a referent is described, the actual node name (e.g., REF1) is printed first, followed by the referential form.

When we install and test this rule, we get the following fancier output, which is shown here for a series of two sentences so that we can see both the verbose and terse forms of the criticism. The terse option means that the verbose form of the message appears the first time, and the terse form on later times. The philosophy of this is that the verbose message can be more explanatory to remind the user of what the criticism means and what to do about it, and then later appearances of the message can then be more compact.

```
(criticize)

Selected output options are:
Output messages are TERSE and CRITICAL-ONLY
...
(the ship is important \.)
...
WEAK-SENTENCE
The main clause of this sentence simply says that REF1 SHIP is important,
Can you make the sentence more to the point?
...
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

Enter sentence or STOP:
(the electrician is also important \.)
...

WEAK-SENTENCE
Try to say something more to the point about REF2 ELECTRICIAN.
...
```

```
---------------------------------------------------------------------------
```

Notice that we now have a message that is perhaps a little more informative in that it identifies the subject of the sentence to the user. In very elaborate sentences this might help make the message more precise and persuasive. For example:

```
(the job performed by the electrician during shipyard maintenance
that might be required by battle damage is always extremely important
to the effectiveness of the ship in combat situations \.)
...

WEAK-SENTENCE
Try to say something more to the point about REF3 JOB.
```

3.5 Recognizing Possible Procedural Content

A multiple-sentence criticism. The purpose of this example is to illustrate how to construct a complicated criticism. The new criticism is intended to recognize when prose text should be presented as a step-by-step procedure. The criticism is that if there are multiple imperative sentences within a paragraph, then perhaps the paragraph should be written as a step-by-step procedure. We will develop this criticism piecemeal.

Eventually we will need to know whether the imperative sentences are in the same paragraph or not, but first we need to just recognize multiple imperative sentences. So first let's see if the parser and semantics module already give us some markers for imperative sentences that we can use in the new criticism rule. We do this by simply giving some sample sentences to the system and examining the output of the semantics module. We can do this with the convenience function (PROPOSITIONS).

```
(propositions)

...
Enter sentence or STOP:
(first \, press the red button \.)

(FIRST |,| PRESS THE RED BUTTON \. >END-OF-SENTENCE)
Parsing Transitions: 477
((IMPERATIVE-STATEMENT ((ADV FIRST) |,|)
  ((VERB-INF PRESS) ((DEFDET THE) (ADJ RED) (NOUN BUTTON))))
 \.)
(PARSER::SUCCESS
 ($START
  ($STATEMENT
   ($STATEMENT-SIMPLE
    ($IMPERATIVE-STATEMENT ($INITIAL-VERBMOD (ADV FIRST) |,|)
                           ($IMPERATIVE-VERBPHR ($VERB-INF (VERB-INF PRESS))
                            ($VCOMP-ACTIVE
                             ($NP
                              ($NP-SIMPLE (DEFDET THE)
                                          ($ADJ ($ADJ-SIMPLE (ADJ RED)))
                                          (NOUN BUTTON))))))
   \.)
  >END-OF-SENTENCE))
Pronoun antecedents:
NIL
ACT Propositions:
((S PROP3 THE_READER) (P PROP3 PRED1) (R PRED1 PRESS) (A PRED1 REF1) (S PROP1 REF1)
 (P PROP1 RED) (S PROP2 REF1) (P PROP2 BUTTON) (S PROP4 PRED1) (P PROP4 FIRST))

ALL Propositions:
((TAG PROP3 SENTENCE-MAIN) (TAG SELF STATEMENT-SUBJECT) (S PROP3 SELF)
 (P PROP3 PRED1) (R PRED1 PRESS) (A PRED1 REF1) (TAG REF1 COMPLETELY-NEW)
 (TAG REF1 NEW-REFERENT) (TAG REF1 NP-REFERENT) (TAG PROP1 NP-PROP)
 (S PROP1 REF1) (P PROP1 RED) (TAG PROP2 NP-PROP) (S PROP2 REF1)
 (P PROP2 BUTTON) (TAG PROP2 HEAD-NOUN) (TAG REF1 DEFINITE)
 (NUMBER REF1 SINGULAR) (GENDER REF1 N) (SRF REF1 (RED BUTTON))
 (HNF REF1 BUTTON) (NP-PROP-NUMBER REF1 2) (S PROP4 PRED1) (P PROP4 FIRST)
 (TAG PROP3 IMPERATIVE) (TAG PROP3 STATEMENT-MAIN) (TAG SENTENCE-PROP-NUMBER 4))
```

The ACT propositions in the output show that this imperative sentence is analyzed as the reader (THE_READER) having a PRESS relationship to an item which is RED and is a BUTTON. The syntactic markers are listed in the All Propositions output. Notice that there is a tag that shows that PROP3 is an imperative sentence. So apparently the parser and the semantics module already give us the markers that we need; all we need is a rule that looks for multiple instances of propositions tagged as imperative.

First draft rule. We need to decide what we mean by *multiple* imperative sentences; let's just say if three imperative sentences appear, then we will trigger this criticism. We want the criticism to appear relatively late in the criticism process, where the relationship of the sentence to the rest of the passage is being commented upon. Examining the production rules, we see that a good place to try it will be at (STEP FINAL-REPORTS). At this point the last imperative sentence will still be in sentence memory (SM) but the previous two will be in passage memory (PM). Here is our first draft rule:

```
(MULTIPLE-IMPERATIVES
IF ((STEP FINAL-REPORTS)
    (SM TAG ?P1 IMPERATIVE)
    (PM TAG ?P2 IMPERATIVE)
    (PM TAG ?P3 IMPERATIVE)
    (DIFFERENT ?P2 ?P3)
    (UNIQUE ?P2 ?P3)
)
THEN ((WRITE-MESSAGE POSSIBLE-PROCEDURE CRITICISM
       "Three imperative sentences have appeared.  Should this be a procedure?"
       NIL)
))
```

This rule needs some explanation; some of the more advanced features of PPS appear here. The rule looks for a pattern in which a proposition in sentence memory is tagged as imperative, and two propositions in passage memory are also tagged as imperative. The pattern predicate DIFFERENT requires that ?P2 and ?P3 be matched to different propositions. Without this predicate, the pattern matcher will find a match in which both of these variables are assigned to the same proposition, which of course is not what we want. In addition, the pattern matcher is very general and will give us a match in which ?P2 and ?P3 are both assigned to two propositions, say PROP1 and PROP2, but with both possible assignments. That is, one match would have ?P2 = PROP2 and ?P3 = PROP3, and then another match would have ?P2 = PROP3 and ?P3 = PROP2. Using the UNIQUE pattern predicate requires that only one set of unique assignments be given to ?P2 and ?P3.

The action of the rule then will simply write out the message, called POSSIBLE-PROCEDURE, that three imperative sentences have appeared and asking whether this should be a procedure. Thus, this rule will fire on a match of one imperative sentence in sentence memory and two different ones in passage memory.

However, the rule will only fire if the imperative tag is in passage memory. Normally the tags are thrown away when the semantic content of the sentence is added to PM, since the tags usually show just syntactic information about the sentence. So we need an additional rule to ensure that we keep the imperative tag around. Looking at the production rules, we see that the updating of PM is done at STEP UPDATE-PM. So the following simple rule will copy an imperative tag in SM to PM.

```
;Save an imperative tag
(START-IMPLIED-REFERENT-HNF
IF ((STEP UPDATE-PM)
    (SM TAG ?P IMPERATIVE)
)
THEN ((ADDDB (PM TAG ?P IMPERATIVE))
))
```

Now that we have installed our two rules we can then try them out using the criticize function. We will type in a series of simple imperative sentences and see if our rule fires.

```
...
> (criticize)

Enter sentence or STOP:
(press the red button \.)

...

Enter sentence or STOP:
(press the blue button \.)

...

Enter sentence or STOP:
(press the green button \.)

...

POSSIBLE-PROCEDURE
Three imperative sentences have appeared.  Should this be a procedure?
```

Sure enough, the criticism appears after we have typed in three imperative sentences. But when we continue and type in a fourth sentence, *Press the yellow button,* the criticism appears again; moreover it appears twice:

```
Enter sentence or STOP:
(press the yellow button \.)

...

POSSIBLE-PROCEDURE
Three imperative sentences have appeared.  Should this be a procedure?

POSSIBLE-PROCEDURE
Three imperative sentences have appeared.  Should this be a procedure?
```

The lockout technique. Let's say we do not want the criticism to reappear, although it might be wise to keep pestering the writer. But we certainly do not want the criticism appearing multiple times. Why is this happening? The reason why is that we now have a total of four imperative sentences in CCS, three in PM and one in SM, and there are thus *two* combinations of these sentences in PM (the first and second, and the first and third) that, together with the fourth sentence in SM, match the pattern in our MULTIPLE-IMPERATIVES production rule. So the rule is fired twice, once for each combination of variable values. If we went on to add another sentence we would get even more repetitions of the message. We will prevent the repetitions by using the *lockout* technique. We will also modify the wording of the message a little bit as well.

Something to remember about PPS is that unlike many other production rule interpreters, PPS is written to be very simple and very general. Thus it does not keep track of how many ways a production rule can fired now or whether it has been fired before. Rather any rule whose conditions are met will fire at any time. So a simple technique to ensure that a production rule fires only once is to have it add a piece of information to the database whose absence is tested in the condition:

```
(MULTIPLE-IMPERATIVES
IF ((STEP FINAL-REPORTS)
    (SM TAG ?P1 IMPERATIVE)
    (PM TAG ?P2 IMPERATIVE)
    (PM TAG ?P3 IMPERATIVE)
    (DIFFERENT ?P2 ?P3)
    (UNIQUE ?P2 ?P3)
    (NOT (PM POSSIBLE-PROCEDURE))
)
THEN ((ADDDB (PM POSSIBLE-PROCEDURE))
      (WRITE-MESSAGE POSSIBLE-PROCEDURE CRITICISM
       "Three imperative sentences have appeared in this paragraph.~
       ~%Should it be written as a step-by-step procedure?"
       NIL)
))
```

The action of this rule adds to the PM database the symbol POSSIBLE-PROCEDURE. The condition clause of the rule with the NOT function tests for the absence of this symbol in PM. Thus, the rule will only fire if POSSIBLE-

PROCEDURE is not in PM. When the rule fires, it adds POSSIBLE-PROCEDURE to PM, and so will not fire again.

To save time when testing this revision, we will use the CRITICIZE-FILE function and put a sample test in a file called "test:"

```
Press the red button.  Press the green button.  Press the blue button. Press the
yellow button. Open the valve. Close the door.
```

Here in condensed form is the results of running CRITICIZE-FILE on this file:

```
(criticize-file "test")
...
Press the red button.

...
Press the green button.
...

Press the blue button.
...

POSSIBLE-PROCEDURE
Three imperative sentences have appeared.  Should this paragraph be
a step-by-step procedure?
...

Press the yellow button.
...

Open the valve.
...

Close the door.
...

Processed 6 sentences.
```

So we see that the POSSIBLE-PROCEDURE message appears as soon as we have entered three imperative statements, but it stays locked out for the rest of the passage.

Paragraph sensitivity. We want this criticism to apply to individual paragraphs. If the text is already written as a step-by-step procedure, then presumably each step will have been formatted as a separate paragraph, and so the criticism would not be triggered. So we will arrange to clean up and reset the contents of PM whenever we start a new paragraph.

The processing of paragraphs is part of the STEP CRITICIZE-TOPICALIZATION phase. This is where we process a new paragraph and we can cleanup there. If the first sentence of the paragraph happens to be an imperative sentence that tag will be copied into PM during the later UPDATE-PM step. We will try adding the following two rules to clear the imperative tags at the beginning of a paragraph and also to remove the POSSIBLE-PROCEDURE information when we start a new paragraph. Note that we have to do this in two separate rules, since we may have imperative tags present but not have triggered the POSSIBLE-PROCEDURE rule:

```
(CLEAR-IMPERATIVE-TAGS-WHEN-PARAGRAPH
IF ((STEP CRITICIZE TOPICALIZATION)
    (SM TAG ??? >PARAGRAPH)
    (PM TAG ?P IMPERATIVE)
 )
THEN
   ((DELDB (PM TAG ?P IMPERATIVE))
))

(CLEAR-POSSIBLE-PROCEDURE-WHEN-PARAGRAPH
IF ((STEP CRITICIZE TOPICALIZATION)
    (PM POSSIBLE-PROCEDURE)
 )
THEN
   ((DELDB (PM POSSIBLE-PROCEDURE))
))
```

The first rule detects the combination of something tagged as a paragraph; this is how the parser and semantics module process a text-formatting command designating the beginning of a new paragraph.   The rule simply removes any and all imperative tags in PM. The second rule simply matches the POSSIBLE-PROCEDURE clause and removes it.

A new test file will make it easy to try this out. The first paragraph will have just one imperative sentence in it; the second will have several and the third a couple of imperative sentences, and so forth. This way we can exercise our new criticism rule:

```
.pp
The maintenance procedure must be performed correctly.
.pp
Press the red button.  Press the green button.  Press
the blue button. Press the yellow button.

.pp
Open the valve. Close the door.

.pp
Close the valve. Open the door. Shift the strainer.
Clean the filter.

.pp

The maintenance is now complete. Report to the duty officer.
```

The following is a condensed form of the output. Here we see that the new criticism is working correctly; the second paragraph triggers the POSSIBLE-PROCEDURE criticism as does the fourth paragraph, and no others.

```
.pp
The maintenance procedure must be performed correctly.
...
.pp
Press the red button.
...
Press the green button.
...
Press the blue button.
...
POSSIBLE-PROCEDURE
Three imperative sentences have appeared in this paragraph.
Should it be written as a step-by-step procedure?
...
Press the yellow button.
...
.pp
Open the valve.
...
Close the door.
...
.pp
Close the valve.
...
Open the door.
...
Shift the strainer.
...
POSSIBLE-PROCEDURE
Three imperative sentences have appeared in this paragraph.
Should it be written as a step-by-step procedure?
...
Clean the filter.
...
.pp
The maintenance is now complete.
...
Report to the duty officer.
...
Processed 13 sentences.
```

Conclusion. Thus, this example shows not only how to add a specific rule criticism that delivers a message, but also rules make use of passage memory to keep track of information over multiple sentences, and how the rules can make use of the text-formatting commands in the input.

4. CRITICISMS CURRENTLY PROVIDED BY CCS

This listing is divided into sections corresponding to the STEP clauses added and removed by the production rules. Each criticism is identified with the output message tag, and briefly described. The actual messages produced are then listed. Messages identified with "V:" and "T:" are the verbose and terse message forms. If no verbose or terse forms are identified, the same message is used for both.

4.1 Gramatical and Sentence-Level Criticisms

```
(STEP CRITICIZE SENTENCE-LEVEL)
```

NONPARSABLE

Detect the nonparsable tag (SM TAG SENTENCE NONPARSABLE) that is produced by the semantics module $GRAMMAR-PROBLEM to designate a grammar problem. A complete parse failure is reported by ATTEMPT-PARSE in "ccs.funcs".

```
V:      Couldn't figure out sentence grammar, but trying to continue with the analysis.
        Some of the following comments may be inappropriate.
        The sentence may be telegraphic in style, too complex, or unusual in
        structure or punctuation - try to simplify the sentence.

T:      Can't analyze grammar; try to simplify sentence; attempting to continue.)
```

OBJECT-RELATIVE-CLAUSE

The sentence includes a clause whose proposition is tagged as an object relative clause (e.g.,*the engine powers* in *The ship that the engine powers is large*) and is also tagged as including a relative pronoun (*that*). The logical subject (*engine*) is described as connected to the logical object (*ship*) by the verb or relation (*powers*).

```
V:       The clause describing these new items is an 'object relative clause:'
         <logical subject> as connected by <relation> to <logical object>
         This construction can be hard to understand - can you rewrite it?

T:       Try to rewrite clause about <logical subject> connected by <relation> to <logical object>
```

OBJECT-RELATIVE-NO-PRONOUN

Like OBJECT-RELATIVE-CLAUSE, but without a relative pronoun present. The grammar may not allow an object relative clause without the relative pronoun.

```
V:       The clause describing these new items is an 'object relative clause:'
         <logical subject> as connected by <relation> to <logical object>
         Because the relative pronoun 'that' is missing, it is very hard to understand.
         Add a 'that', or even better, rewrite the clause.

T:       Add Pronoun that or rewrite clause about about <logical subject> connected by <relation> to <logical object>
```

CENTER-EMBEDDED

The classic impediment: *The oil that the pump that the motor powers circulates lubricates the bearings.* This rule detects the presence of two object relative clauses that are chained together. The three referents (*motor, pump, oil*) are listed to identify the clauses.

```
V:       Ouch! These new items appear in a 'center embedded' structure:
         <list of referents>
         This construction is extremely hard to understand and should be rewritten.
```

```
        Try rearranging the ideas into a simpler order.

T:      Ouch! Rewrite clause about <list of referents>
```

MISSING-ARTICLES

These are accumulated and reported at the end of this STEP. Noun phrases which have no article or determiner, and have a head noun which is not marked as MASS in the lexicon (e.g. *water, sand*), are reported.

```
V:       Possibly missing articles (a, the) for the following:
         <referent name>
         Telegraphic style is hard to read and should be avoided.

T:       Check and write in if needed:
```

NEGATED-MAIN-CLAUSE

Criticizes a statement based on a negation - a clause that is tagged as the main clause of a sentence, and also has the predicate FALSE attached to it (e.g. *The ship is not large* .)

```
V:      The main clause of this sentence is negative, which can be hard to understand.
        Can you rewrite the sentence into a positive form?

T       Try to rewrite into positive form.
```

WEAK-SENTENCE

An experimental rule. If the main clause of the sentence is simply a predication that the statement subject is important, criticize as being weak. The subject is listed.

```
V:      The main clause of this sentence simply says that <subject> is important,
        Can you make the sentence more to the point?

T:      Try to say something more to the point about <subject>.
```

MULTIPLE-NEGATION

Like NEGATED-MAIN-CLAUSE, but comments on a sentence that has two or more negated propositions, which may not include the main clause. This message could appear more than once if there are more than two negated propositions.

```
V:      This sentence has multiple negatives, which can be very hard to understand.
        You should rewrite it into positive form.

T:      Try to rewrite into positive form.
```

SENTENCE-TOO-BIG

The semantics module counts how many propositions remain in the sentence memory representation after propositions describing given referents have been removed, and supplies this information in SM to the criticisms rules. This is a rough indication of how much new information that the reader must assimilate in this sentence. A threshold of 12 propositions is used; this can be easily changed. This number is based on what intuitively seems to be too much information. The literature on the role of Short-Term Memory capacity during comprehension is incomplete.

```
V:      The sentence contains <number> propositions of new information, which may be too
        much to understand easily.  Check whether all of the information is actually
        necessary at this point, and if so, try expressing it with smaller sentences.

T:      <number> propositions of new information - simplify or break up.
```

DESCRIPTIONS-TOO-BIG

This is accumulated and reported at the end of this phase. Noun phrases that describe new referents using more than some number (currently 7) of propositions are reported.

```
V:        These items were described in noun phrases that are probably too large:
          <referent name> was described with <number> propositions
          Try to refer to them with smaller, more compact descriptions.

T:        Try smaller descriptions for:
```

NONPARSABLE-SENTENCE-SUBJECT - INFORMATION

If the sentence is nonparsable, the semantics module guesses the subject as being the first noun phrase. This message reports it.

```
V:       Can't be sure of the sentence subject,  but assuming that it is <subject>
         Later comments may be inappropriate if this is not the correct subject.

T:       Assuming subject is <subject>.
```

SENTENCE-SUBJECT-ACTIVE and SENTENCE-SUBJECT-PASSIVE - INFORMATION

These comments report the main subject of the sentence; the corresponding rules identify the subject for the remainder of the criticism rules.

```
       The main subject of this sentence is <subject>
```

MAIN-PROP-RELATION - INFORMATION

This comment points out the main proposition in the sentence.

```
       The main proposition of this sentence is <proposition name>
        - <logical subject> has relation <relation> to <logical object>
```

MAIN-PROP-PREDICATE - INFORMATION

This comment points out the main proposition in a copulative sentence.

```
      The main proposition of this sentence is <proposition name>
      <subject> is modified by <predicate>
```

4.2 Referential Criticisms

```
(STEP ANALYZE REFERENCE)
(STEP CRITICIZE REFERENCE)
```

AMBIGUOUS-REFERENT

This message appears if the reference module tags a reference noun as ambiguous; the most recent one is assumed, and the other possibilities are listed.

```
V:      Reference to these items was ambiguous:
        Assuming that <noun> is <referent>
          ... other possibilities: <list of referents>
        Check, and rephrase the description if this is not correct.

T:      Check and rephrase if incorrect:
```

MATCHED-REFERENT

A referent is tagged by the reference module as a GIVEN-PRED-MATCH, meaning the a referent was referred to using wording that was not the same as previously used, but still sufficed to identify the referent. Thus the reference might be incorrect, or might be using inconsistent terminology.

```
V:        Possible incorrect or inconsistent terminology -
          These items were matched with previously mentioned ones as follows:
          <reference> is previous <referent>
          Check, and correct the terminology, or consider terminology
          that will allow these to be referred to consistently.

T:        Check for consistent terminology:
```

QUESTIONABLE-NEW-REFERENT

A referent is new, but was mentioned in a definite noun phrase, thereby implied that the reader is supposed to know about it. This comment is made only if a sentence (rather than a heading) is present - there must be something tagged as a main proposition.

```
V:        These items were referred to as if the reader already knows about them,
           but they could not be matched with something previously introduced:
          <referent>A
          Check: Can your reader easily figure out what you are referring to?

T:        Check: Can your reader tell what you are referring to:
```

HEADING-NEW-REFERENT - INFORMATION

If a heading contains a new referent, report it.

```
V:        This heading introduced these new items or concepts:
          <referent>
          Check that they are really new and should be introduced at this point.

T:        This heading introduced these new items or concepts:
```

INDEFINITE-NEW-REFERENT - INFORMATION

If a new referent appears in an indefinite noun phrase, or a new referent is a plural head noun with no determiner. This comment is made only if a sentence is present.

```
V:        This sentence introduced these new items:
          <referent>
          Check that they are really new and should be introduced at this point.

 T:       This sentence introduced these new items:
```

PRONOUN-REFERENCE

This message describes the assumed antecedent for a pronoun; the system has no way to tell what the pronoun antecedent is supposed to be, so this informative comment is the only possible critique. Only third person pronouns are commented upon.

```
V:       Readers will tend to assume that the pronoun <pronoun>
         refers to <antecedent>
         If this is incorrect, avoiding use of the pronoun might be more clear.

T:       Pronoun <pronoun> should refer to <antecedent>; avoid pronoun if not.
```

UNKNOWN-PRONOUN

This comment appears if there is no apparent antecedent for a pronoun; a dummy item, UNKNOWN, is used instead

```
         Can't tell what pronoun <pronoun> refers to - using UNKNOWN
         Rewrite so that either no pronoun is used, or it has a clear antecedent.
```

NO-KNOWN-REFERENTS

This criticism is made if the sentence contains only new referents - this means that the sentence does not have any clear or direct relationship to the preceding passage sentences.

```
V:     This sentence does not appear to refer to anything previously mentioned,
       and so readers may not understand how it relates to the rest of the material.
       Be sure that the sentence directly and clearly refers to a previous item.

T:     Sentence should clearly refer to something previously mentioned.
```

SUBJECT-NOT-KNOWN

If the subject of a statement is a new referent, this criticism appears.

```
V:      The clause subject <subject>
        could not be matched with a previously mentioned item.
        Try to make the clause subject be a previously mentioned item.

T:      Clause subject <subject> should be previously mentioned.
```

COMPLICATED-INTRODUCTION

This criticism appears if a new referent does not have a simple referential form. The idea is that a new object should be introduced with a simple noun phrase, one that does not have relative clause modifiers.

```
          These items were first mentioned in a complicated form:
          <referent>
          Try introducing them is a simpler way, without modifying phrases.

          Try simpler introduction of:
```

NO-SIMPLE-FORM

This criticism appears if a given referent did not have a simple referential form before, but does in this sentence.

```
V:       These sentence items were originally described in a more complicated way:
         <referent>
         Can you introduce them in the simpler wording in this sentence?

T:       Try introducing these with the simpler wording in this sentence:
```

IMPLIED-REFERENT - INFORMATION

In phrases like the wing of the airplane, in which one noun phrase is attached by the relation OF to another noun phrase. This rule comments that if a new noun phrase is so attached to another, the new reference is treated as being implied by the other. E.g., if we have an airplane, we have anything that is OF an airplane.

```
V:       Assuming that these newly introduced items are implied by other items:
         New <referent> is implied by <other referent>
         Check: Does the reader know these implications?

T:       Check that the reader knows these implications:
```

4.3 Topic Structure Criticisms

```
(STEP CRITICIZE TOPICALIZATION)
```

There is a set of rules that build a topic structure, and nominate various things as possible topics. Some of these rules generate only INFORMATION messages, but they also identify and construct the topic structure.

The LOCAL TOPIC is updated when the sentence is criticized, which is also when the local topic changes are commented on. The rules assume that PM initially contains a dummy skeleton of topic structure so that there are dummy points to represent missing levels of the structure.

TITLE-GLOBAL-TOPIC - INFORMATION

A global topic specified by a title (>TITLE) is reported.

```
Global topic is <referent>
```

HEADING-GLOBAL-TOPIC - INFORMATION

A global topic specified by the first heading is reported.

```
Global topic is <referent>
```

HEADING-SUBTOPIC - INFORMATION

Report a subtopic, possibly specified by >TOPIC-HEADING, of the current global topic

```
New subtopic of <referent> is <referent>
```

HEADING-PARAGRAPH-TOPIC - INFORMATION

Report the topic of a paragraph heading.

```
       Heading introduces paragraph topic
       <referent> as a subtopic of <referent>
```

TOPIC-SENTENCE - INFORMATION

Report the subject of a topic sentence as a subtopic.

```
      Paragraph topic sentence introduces <referent>
       as a subtopic of <referent>
```

PASSIVE-OK

Detect and comment upon a use of the passive which is appropriate because it makes the surface subject the same as a current topic.

```
V:    Using the passive voice in the clause about <referent>
      is OK because it is about the current topic.
      But check to see if you can rephrase the clause as active.

T:    Can you rephrase passive clause about <referent>
```

CHAIN-PASSIVE-OK

Detect and comment upon a use of the passive which is appropriate because it makes the surface subject the same as the object of the previous sentence (a chain construction).

```
V:    Using the passive voice in the clause about <referent>
      is OK because it is about an item in the previous sentence.
      But check to see if you can rephrase the clause as active.

T;    Can you rephrase passive clause about <referent>
```

INAPPROPRIATE-PASSIVE

Detect and criticize an inappropriate use of the passive.

```
V:    Using the passive voice in the clause about <referent>
      is inappropriate because it is not about a current topic.
      This can be very hard to understand - try to rephrase into the active voice.

T:    Rephrase passive clause about <referent>
```

SUBJECT-STAYS-ON-TOPIC - INFORMATION

Comment on a simple form of coherence - the sentence subject is also the current topic.

```
      Sentence stays on the current topic <referent>
```

CHAIN-TOPIC-CHANGE - INFORMATION

Comment on how a chained sentence construction coherently changes the topic - the current sentence subject is the object of the previous sentence.

```
        'Chained' sentence changes the current topic
        from <referent> to <referent>
```

NEW-SUBJECT-CHANGES-TOPIC

Criticize how using a new referent as the sentence subject produces incoherence.

```
V;     Possible incoherence: This sentence changes the current topic
       from <referent> to <referent>
       which has not been mentioned before. You may need to rewrite
       to achieve a good transition.

T:     You may need to rewrite to avoid incoherence.
```

TOPIC-CHANGE

Criticize a milder form of incoherence; the sentence subject is neither the subject nor object of the previous sentence, but is already known.

```
V:     Possible incoherence: This sentence changes the topic
       from <referent> to <referent>
       which was not mentioned in the previous sentence.

T:     You may need to rewrite to avoid incoherence.
```

SET-LOCAL-TOPIC - INFORMATION

The current sentence subject is added as a local topic, and is made the current local topic.

```
       Sentence subject <referent> is
       now a local topic under <referent>
```

4.4 Make Final Reports

```
(STEP FINAL-REPORTS)
```

CURRENT-LOCAL-TOPIC - INFORMATION

Report the current local topic.

```
       Current local topic is <referent>
```

POSSIBLE-PROCEDURE

This is an experimental criticism; it is triggered by the appearance of three imperative sentences within a single paragraph, which suggests the presence of procedural content, expressed as a prose paragraph.

```
      Three imperative sentences have appeared in this paragraph.
      Should it be written as a step-by-step procedure?
```

REFERENCES

Allen, J. (1987). *Natural language understanding.* Menlo Park, CA: Benjamin/Cummings.

Anderson, J. R. (1976). *Language, memory, and thought*. Hillsdale, NJ: Lawrence Erlbaum Associates.

Covrigaru, A., & Kieras, D. E. (1987). *PPS: A Parsimonious Production System* (Tech. Rep. No. 26). (TR-87/ONR-26). Ann Arbor: University of Michigan, Technical Communication Program.

Forgy, C. L. (1982). Rete: A fast algorithm for the many pattern/many object pattern match problem. *Artificial Intelligence*, **19**, 17-37.

Kieras, D. E. (1977). Problems of reference in text comprehension. In M. Just, & P. Carpenter (Eds.), *Cognitive processes in comprehension*. Hillsdale, NJ: Lawrence Erlbaum Associates.

Kieras, D. E. (1983). A simulation model for the comprehension of technical prose. In G. H. Bower (Ed.), *The Psychology of Learning and Motivation*, **17**. New York, NY: Academic Press.

Kieras, D. E. (1985a). *The potential for advanced computerized aids for comprehensible writing of technical documents* (Tech. Rep. No. 17, TR-85/ONR-17). Ann Arbor: University of Michigan, Technical Communication Program.

Kieras, D. E. (1985b). *Improving the comprehensibility of a simulated technical manual* (Tech. Rep. No. 20, TR-85/ONR-20). Ann Arbor: University of Michigan, Technical Communication Program.

Kieras, D.E. (1987). *A computerized comprehensible writing aid: Final report* (Tech. Rep. No. 27, FR–87/ONR–27). Ann Arbor: University of Michigan, Technical Communication Program.

Kieras, D.E., (1989). An advanced computerized aid the for writing of comprehensible technical documents. In B. Britton & S. Glynn (Eds.), *Computer Writing Aids: Theory, Research and Design*. Hillsdale, NJ: Lawrence Erlbaum Associates.

Mayer, J., & Kieras, D. E. (1987). *A development system for augmented transition network grammars and a large grammar for technical prose* (Tech. Rep. No. 25, TR-87/ONR-25). Ann Arbor: University of Michigan, Technical Communication Program.

Steele, G. L. (1984). *COMMON LISP: The language.* Bedford, MA: Digital Press.

Winograd, T. (1983). *Language as a cognitive process.* Reading, MA: Addison-Wesley.