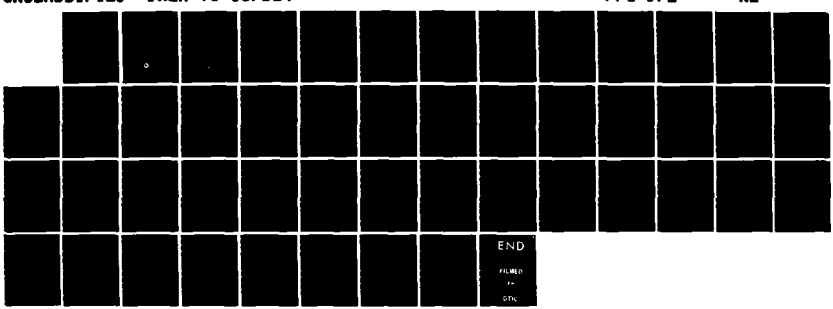
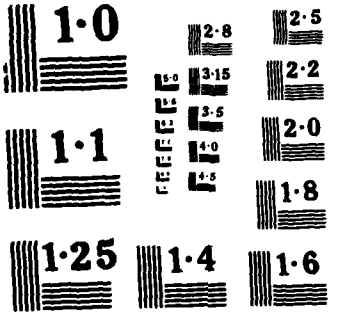
AD-A163 279 CIRCE A NATURAL LANGUAGE RULE TRANSLATION SYSTEM FOR INTERSENSOR(U) DEFENCE RESEARCH ESTABLISHMENT ATLANTIC DARTMOUTH (NOVA SCOTIA) J R ELLIS ET AL. NOV 85 1/1

UNCLASSIFIED DREA-TC-85/314 F/G 9/2 NL





(3)

UNLIMITED DISTRIBUTION

National Defence
Research and Development Branch

Défense Nationale
Bureau de Recherche et Développment

TECHNICAL COMMUNICATION 85/314
November 1985

CIRCE: A NATURAL LANGUAGE RULE TRANSLATION SYSTEM FOR INTERSENSOR

James R. Ellis - C. Ann Dent

Defence Research Establishment Atlantic

Centre de Recherches pour la Défense Atlantique

Canada

UNLIMITED DISTRIBUTION

National Defence
Research and Development Branch

Défense Nationale
Bureau de Recherche et Développment

CIRCE: A NATURAL LANGUAGE RULE TRANSLATION SYSTEM FOR INTERSENSOR

James R. Ellis - C. Ann Dent

November 1985

Approved by C.W. Bright H/Signal Processing Section

DISTRIBUTION APPROVED BY

D/UAD

TECHNICAL COMMUNICATION 85/314

Defence Research Establishment Atlantic

Centre de Recherches pour la Défense Atlantique

Canada

Abstract

Circe is a set of functions written in LISP as an experiment in natural language processing. It was developed for knowledge acquisition by the Intersensor system, a sonar knowledge-based inference system. The main function of Circe inputs from either the keyboard or a file a rule written in English and converts it into a LISP-like data structure. This conversion takes place in two steps: the English text is first parsed, and then the relevant information is extracted from the resulting list and assembled into rules. These rules are then written to a file where they can be used directly by the inference mechanism of Intersensor without further processing.

The paper briefly reviews prior research at DREA in this area, and contrasts the current system with its predecessors. Implementation problems are discussed, and suggestions for improvements are given.

RESUME

Circe est un ensemble de fonctions rédigées en langage Lisp à titre d'expérience de traitement en langage naturel. Il doit permettre l'acquisition de connaissances à l'aide du système Intersensor, système d'inférence exploitant une base de connaissances établie à l'aide d'un sonar. La fonction principale de Circe introduit à partir d'un clavier ou d'un fichier une règle rédigée en anglais et la convertit en une structure de données se rapprochant d'une structure Lisp. Cette conversion se fait en deux étapes: le texte anglais fait d'abord l'objet d'une analyse syntaxique; ensuite, l'information pertinente est extraite de la liste résultante et transformée en un ensemble de règles. Ces dernières sont ensuite versées à un fichier où elles peuvent être exploitées directement par les dispositifs d'inférence d'Intersensor.

Dans le document, on analyse brièvement la recherche effectuée antérieurement au Centre de recherches pour la défense l'Atlantique (CRDA) et on fait le pendant entre le système courant et ceux qui ont été exploités antérieurement. Les problèmes de mise en place y sont analysés et on y fait des suggestions visant à améliorer le système.

Contents

1 Introduction

Circe is an experiment in natural language understanding and processing. It was designed to be a development aid for **Intersensor**, a knowledge-based inference system. (See [6] for more on **Intersensor**) **Circe** was developed to allow a sonar expert with minimal computer expertise to enter rules for the system in English. **Circe** attempts to convert the English rule into LISP code that can be used directly by **Intersensor**. This conversion takes place in two steps; the rule is first broken down into its component parts of speech and then this parse is analyzed to extract and organize the relevant information.

There has been other research carried out at DREA with the same goal as **Circe**. In 1979, Mark Rodger [9] wrote a context-free parser using Interlisp, for use on the DEC-20 system. The program allowed the user to enter his own grammar into the system, and to specify whether the parser was to be goal or event driven (top-down parsing as opposed to bottom-up). The program has no apparent connection to **Intersensor**, and went no further than presenting the results of the parse to the user. **Calliope I** [7] , developed by Jennifer Muise, was a top-down, left-to-right parser. It was intended as a first step towards a larger system that would satisfy five goals:

1. Accept English input and have a basic understanding of the words used.
2. Parse the input successfully.
3. Convert the English rules into LISP code.
4. Make the code accessible to **Intersensor**.
5. Provide an explanation facility for the rules (similar to the query answering system for **MYCIN** [3]).

The original **Calliope** only went as far as the second goal. It parsed successfully a number of test rules according to a reasonably complex grammar.

The second **Calliope** [8] accomplished all but the last goal. Like **Calliope I** , **Calliope II** 's grammar was implemented using units in the **Athena** knowledge representation language [5] . A unit in **Athena** is a frame-like data structure. For **Calliope**, the units contained the name of a part of speech and some information about it. **Calliope II** could parse a rule using a slightly revised version of **Calliope I** 's parser, and then insert it into **Intersensor**'s knowledge base using **Athena**. Unfortunately its grammar was by necessity very limited. In contrast, **Circe** works with a grammar that is larger than that of the original **Calliope**, and translates successfully most of the rules that can be formed using its grammar. Unlike **Calliope** , **Circe** does not attempt any interface with either **Athena**, or **Intersensor**.

Chapter II describes a session of Circe. Chapter III and IV discusses design and implementation details of parser and translater respectively. Chapter V suggests improvements for Circe while Chapter VI contrasts the current system with its predecessors at DREA.

2 Using Circe

Circe has two major components: the parser and the translator. To run the program the user must first load the file containing the translator (at the time of writing Bl:>jim->circe-translator.bin[1]) and then the file containing the parser and the top-level functions. (Bl:>jim>circe-parser.bin). Once the files are in the workspace, the user can start the process by calling the function Circe. The function begins by setting the initial conditions and getting the first input string. It calls the parser (**Perform-test**), gives the result to the translator (**Make-rule**), and then displays both the resultant LISP code and a translation of that code back into English. Processing continues with one rule input each loop, until the user signals that he is finished; at this point Circe prompts for the name of a file in which to write the rules and the program ends after the writing has been completed.

Immediately after invoking Circe the user is presented with a menu with which he can change the input and output conditions of the program. The user can specify either terminal or file input, and whether or not he wishes some intermediate output the program can provide. He can also change the default file that the program will read from if file input is chosen. It is important that each rule in the input file specified is followed by a carriage return, as this is the delimiter Circe recognizes for end of sentence. If the user opts to process rules from a file, the program will continue using this method until directed otherwise, although an exception to this occurs when a rule is not processed to the user's satisfaction. In this case the program prompts the user to re-enter the rules via the keyboard. The old rule can be retrieved for editing at this point by the meta-control-y key sequence. Once the user indicates the rule is satisfactory, Circe goes back to the file to get the next rule.

The intermediate output that can be requested in the initial menu are tracing messages and the results of the parser. The result of the parse is a very long list containing a list for each simple statement in the rule, which in turn is a list of all the parts of speech identified in that simple statement. This is longer than might be expected. Longer still are the tracing messages, which the user should almost never have the need to see. The tracing messages give a running account of where the parser has gone in the parse tree, giving the name of every node it tries and whether or not each test was successful. Unless something appears to be drastically wrong with the parse, the user should probably set both of these variables to "No."

After the input conditions are set, the main loop of Circe begins. Each loop processes completely one rule. If the user has opted to type in the rules, he will be prompted at the beginning of the loop to enter a rule, followed by a carriage return. It is important not to hit the carriage return before the rule is completely typed in because processing starts immediately after the carriage return is entered. The user can exit the program at this point by typing the word "stop," or change the input/output conditions by typing the word "options." If the rules

[1]All filenames, control-keys and other implementation-specific items are in reference to a Symbolics 3600 workstation

are being read from a file, the user is asked if he wants to process a rule from the file. If he answers no, he will be asked if he wishes to change the options or stop.

Once **Circe** has a rule, it calls the method **Perform-test** of the flavour *Sentence-parts*, which does the actual parsing. As each word of the input sentence is encountered and identified, it is displayed on the screen. If a word cannot be identified, a message is given to the user telling which word caused the error and one of the parts of speech the word could have been. It is impossible to tell exactly which part of speech the user meant there to be at that point, because usually several different parts of speech may be correct at any given point in a sentence. (See Appendix A for a definition of the current grammar or Appendix B for its graphical representation.) Occasionally the program may even get offending word wrong, if the program happened to back up in the parse to try to correct the mistake. If the user has entered a word not in **Circe**'s vocabulary, then the appropriate vocabulary list will have to be updated to include this word, if indeed it should include it. (See Appendix F for the method of updating the vocabulary.)

If the parse is successful, the program will display the result if the user has requested this, and will call the translation routine **Make-rule**. The resulting LISP code will be displayed, as will the rule translated from LISP back into English. The user should carefully examine these to be certain that the rule says exactly what he wanted it to say. The program will ask the user if the rule has been processed correctly; if it has not, **Circe** will prompt the user to re-enter the rule. The user must rephrase the part of the rule that was incorrectly interpreted. Re-entering the same wording will give the same result as the previous parse. This question does not result in a call to an editing facility; it merely stops the incorrect rule from being saved. The corrected sentence is treated exactly as if it were a new sentence.

Should the user reply that the rule was correctly processed, the program prompts for an identifier for the rule. The identifier, the input, and the LISP rule are then put on a list as a single unit. The main loop is now finished, and **Circe** again prompts for input. If the current mode of input is file, and the current file is empty, the user will be informed and asked whether he wants to change the input conditions or end.

At the end of the session the program requests the name of a file in which to write the rules. This given, it takes the rules off the list of rules one by one and writes them to the file specified. (See Appendix E for some sample output rules.) A message is displayed when the rules have all been written.

3 The Parser

The parser was designed to be as modular as possible. Modularity is desirable in almost any program, because it ensures that a change in one function will not necessitate a change in every function even remotely related to it. It also means that functions need not be duplicated if they are required in two different places. Modularity was especially desirable in this project because the program is an experimental system in a changing environment. If the requirements of the larger system change it should not be too difficult to change the parser to accommodate this. Because the parser is separate from the grammar it uses, any grammar could be substituted for the current one and presumably the system would still function successfully. Similarly, any one of the tests the parser uses could be substituted by another of the same name without altering anything else in the system (providing the number of variables passed to it did not change). (See Appendix F for the methods to change the current grammar and to add a new parsing test.)

The parser uses the recursive properties of Zetalisp to perform a top-down left-to- right search. This search is goal-directed; it starts with one goal, and tries to satisfy that before it moves on to another. Often a goal will have subgoals which must be satisfied, and these in turn will have their own subgoals. The parse is complete when the original goal, that of "statement," has been satisfied.

The grammar and the main parsing function are built on the flavour facility in Zetalisp. (For more information on flavours, see [10]) Each part of speech in the grammar tree is defined as an instance of the flavour *Sentence-parts*. Each instance contains a variable "parts," which is a list of its subgoals, a variable "test," which is a list of tests for these subgoals, and a variable called "significant?" which tells whether or not the node is important enough to include in the parse results. (See Appendix C for examples of this implementation of parts of speech in the current grammar.) Each of these flavor-instances is in effect a Backus Naur form rule (for more information on BNF rules, see [4]). The name of the instance is the non-terminal variable found on the left-hand side of a BNF rule, and the goals on the parts-list form the list of terminal and non-terminal variables on the right-hand side of the rule. The parts-list and the test-list together form the replacement function for the rule.

The parsing function is a method of the flavour *Sentence-parts*. Circe sends a message to the flavour-instance **stmt** to perform the generic function **Perform-test** on itself. **Perform-test** gets the parts list and the test list from **stmt**, and successively applies each test on the list to its corresponding part on the parts-list. Generally these tests consist of sending a message **Perform-test** to the flavour-instance of the part specified, and setting some variables according to the results. When **Perform-test** has applied all the tests to the parts list, it returns a result which indicates whether all of the tests have been satisfied. If they have, then the goal itself has been satisfied and can be entered on the parse-list. (See Appendix D for a graphical example of how the parser works.)

The tests the parser uses can be divided into two types, depending on whether they check for terminal or non-terminal goals. **Man**, **Opt**, **Exor**, **Oob**, and **Double-list** all eventually result in the message **Perform-test** being sent to a part, as they all test for non-terminal goals. **List?**, **Spec?**, **Number?**, and **Pnoun?** check for the existence of a terminal node, so that no call to **Perform-test** is necessary.

All of the first group are, or could have been, built using the one basic function **Man**. **Man** is short for "mandatory"; essentially the function sends a **Perform-test** to the part given to it and returns Nil if the test fails. If it succeeds, **Man** adds an associated pair to the parse-list containing the name of the successful goal and the number of the simple statement in which the goal occurred. If the variable *significant?* of the goal is set to "Nil," the goal name will be associated with "D" rather than the simple statement number. This will cause the pair to be deleted later in the processing. If the goal was satisfied by the existence of a word, the goal is consed with the word itself.

The test **Opt**, short for "optional," could have been written using **Man**, but was not. **Opt** performs basically the same functions as **Man**, but its results are treated differently by the **Perform-test** that called it. This routine is tolerant of failures in an **Opt** test, whereas a failure of a **Man** test signals the failure of the goal. **Opt** also differs from **Man** in that it saves the values of the pointers that tell where the parser currently is in the input sentence. If the subgoals of an **Opt** test fail, the position before the initiation of the test can be restored.

The other functions in the first group - **Exor**, **Oob** and **Double-list** - are all based on the **Man** test. **Exor** performs an exclusive-or test on the list of two parts that is supplied to it. It checks for the first part using **Man** and if successful, it returns without checking for the second. If the first test fails the second part is checked for and if this is unsuccessful "Nil" is returned for the **Exor** test. **Oob** is also supplied a list of two parts. It checks for the existence of one or both of the two parts, in the order they are supplied. Double-list applies the test **Man** successively to a list of items, returning after the first successful test. If all of the lists fail, the test returns "nil."

The second group of tests - **List?**, **Spec?**, **Number?**, and **Pnoun?** - all check for the existance of a certain type of word. **List?** will compare the word in the input string currently being examined against each word in a list of words. For example, if the word being identified was "vessel" and the current goal was a noun, **List?** would have been given N-list as its input variable. **List?** would have compared "vessel" to each word in N-list in an attempt to verify if "vessel" was actually a noun.

Spec? checks for a specific word. At different places in the grammar tree certain words must occur, such as "if" at the beginning of a rule. **Spec?** checks if the word being identified is the same as the word passed to it. **Number?** merely checks whether or not the word being examined is a number (expressed in digits). It has no variables passed to it.

The test **Pnoun?** is not as simple as the other tests in this group. **Pnoun?** decides if the word under consideration is a possesive noun. To do this it checks the list of nouns. Before it can do this, however, it must remove the trailing apostrophe-*s* from the word. Since the words have been converted to symbols to make processing easier, this is impossible. To remedy the situation, the routine that extracts the words from the input string saves any word with an

apostrophe in a global string variable called *Pword*. To check for a possesive noun then, the routine first checks the variable *Pword*. If it is empty, there is obviously no possesive noun. If it is not empty, **Pnoun?** strips off the apostrophe-s, converts the string to a symbol and checks the noun list for its existence. This routine could be made more general purpose by allowing the calling function to specify which word list should be checked. In this way the grammar could conceivably be expanded to handle possessive pronouns and contractions.

All the routines that check for specific words get the next word if the routine successfully identifies the current one. This is done by calling the function **Gettoken**, which extracts the words from the input string using pointers. **Gettoken** sets a pointer to the first nonblank character to mark the start of a word, and a pointer to the next blank after that to mark the end. The pointers leapfrog over each other; each using the other as the starting position for its search. When the two pointers are set the string delimited by them is extracted. Periods are trimmed off the ends of the word and a search is made for apostrophes. If one is found, the string is placed in the global variable pword, and the apostrophe is converted to a number sign. After this the routine checks if the string is a number expressed in digits, in which case it is left as is. Otherwise, the program converts the string into a symbol and places it in the global variable *Word*.

The final output of the parser is a list detailing the results of the search through the grammar tree. The words "if", "premise", "then", and "action" are all on a separate list inside the parse list, as are the elements of each simple statement. Each simple statement list contains a cons for every part of speech that was identified in the statement, in the order in which they occurred. This parse-list still contains all the conses marked with a D; they are eliminated later by the translator.

The output of the parser was originally a hierarchical list that showed more definitely the relationships of the parts within the sentence. Each node was inside a list of each of its ancestors, and nodes with the same ancestor were inside the same list. While it was easier to see the interrelationships within the structure, it was harder to process the structure itself. The simple list structure that was eventually adopted is far easier to work with. The drawback is that there is greater potential for misinterpreting the information contained in the list.

If at some time it should prove more useful to adopt the hierarchical structure again, it is a simple matter of changing the commands in **Perform-test**, **Man**, and **Opt** that deal with values returned from function calls.

4 The Translator

Once the sentence has been broken down into its component parts of speech by the parser, the translator must convert this parse into something more useful. The translator must arrange the parts in the sentence to form a more uniform representation of the knowledge contained in them. The data structure chosen for this is the familiar object-attribute-value triplet. For this application these triplets are augmented by a variable at the beginning to indicate the relation between the attribute and the value, and another variable at the end to indicate the level of confidence attached to the rule. The confidence level is expressed in whole numbers from -100 to 100, depending on the level of certainty required for the rule to be activated. The job of the translator is to take the information in the parse and decide which pieces of information fit where in these pentads.

The input to the translator is the list of lists output by the parser. The first list it ignores; this contains the word If associated with itself, indicating that the sentence lists that follow are in the premise clause. The next list will contain a simple-statement, and the translator can begin the process. The processing by the translator takes the form of a large loop, each loop translating completely one simple statement. For each simple statement the translator will remove any unnecessary information in the list, classify the statement according to sentence-type, and then extract the information necessary to form the relation-object-attribute-value-confidence groups. The first step in this is to remove all the unnecessary associated pairs in the list. The subroutine **Killer** does this by identifying all the pairs with "D" as their cdr and deleting them. The statement is then examined by **Classify** to determine which type of sentence it is. Every sentence can be put into one of seven different categories, depending on such things as the type of verb used and the parts of speech that follow the verb. Each classification is represented by an instance of the flavour *Sentence-Types* that contains a list of the distinguishing characteristics and the type of ending the sentence has.

Having classified the simple statement, the translator begins processing it. The program branches at this point, depending on whether the statement comes from the premise or the conclusion. The program assumes it is working on the premise until informed otherwise. It sets the default conditions for premise, and calls the functions **Subject**, **Predicate**, and **Ending** to extract the information for the relation-object-attribute-value-confidence group.

Subject finds both the object and the attribute of the sentence or sentence-fragment supplied to it. The object of the sentence, if there is one, will either appear as a possessive noun or a noun inside a prepositional phrase. The translator looks for the possessive noun first, then for a noun before a prepositional phrase (which would be the attribute), and finally for the noun in the prepositional phrase. For each noun found **Subject** calls **Extract-Adj** to find any adjectives modifying the noun. A new pentad is created for each adjective located, using IS-SAME as the relation, the noun as the object, the type of adjective as the attribute, the adjective itself as the value and 100 as the confidence level. The translator is able to classify

the adjectives because of the way they are handled by the parser. The adjectives are stored in the parser's vocabulary different lists depending on what type of information they convey. Because the different lists are arranged in a tree under the same ancestor "adjective," the parser creates three conses for each adjective found: one indicating that an adjective exists, the second giving which type of adjective exists, and the third containing the adjective type again and the adjective itself. The second cons is deleted by the translator.

Depending on the type of adjective found, the translator may convert the information to a more uniform representation. If the adjective is a measurement such as "100 feet," the program will look for "feet" in a conversion table and replace it with "ft". It then concatenates this with the number so that the measurement can be treated as a single unit. When the translator finds a measurement it automatically checks for a quality to go along with it, such as "long" or "wide." If it finds one it will try to convert it to a noun, such as "length" or "width," to be used as the attribute element of a pentad. If there is no quality in the sentence, the translator tries to find a default quality for the unit used: the default for "hz" is "frequency." If this also fails, the translator uses the generic "measurement" as the attribute.

Regardless of which adjective is found, it is deleted from the sentence so that it will not be found again by the program. Any pentads that are created for adjectives are placed on a global list to be dealt with later. If the main object and attribute are found they are placed in global variables. These may play an important role in further processing, especially if the next statement contains pronouns. When pronouns are located in a simple statement they are given the value of whichever noun would normally be expected in their position. For example, if a pronoun is found in a prepositional phrase it is given the value of the global variable *Mobj*, which contains the object of the last sentence. If at the end of all this the program has not identified the object or the attribute, the defaults "object" and "attribute" will be used.

Having decided on the first two variables, the translator calls **Predicate** to identify a relation for the group. If the verb of the sentence is an active verb then this becomes the relation. If the verb is a copula verb the translator checks for adverbs that could be used in making the function name of a relational operator. For example, if the copula verb "is" is followed by the adverb "less-than" the function concatenates these to form the function name "IS-LESS-THAN." The program also checks at this point for adverbs that indicate a level of confidence. These adverbs are converted to numbers (see table 1) and placed in the global variable *Mconf*. The default confidence level is always 100 i.e., unless otherwise stated, the program assumes the input to be completely certain.

If the verb is an active-verb the program checks for other types of adverbs, for example ones which would indicate speed or direction. It attempts to convert these to their adjective equivalent, so that "the fast boat" and the boat that "moves quickly" will result in the same representation.

The final step for sentences in the premise is a call to the function **Ending**. **Ending** in turn sends a message to the instance of the flavour *Sentence-Types* that represents the sentence-type of the current statement. The message returns the type of ending the translator should expect to find. The word that fits this description is put into the global variable *Mval*. The routine also checks for a numeric confidence level indicator at the end of the sentence; if found, the number is placed in the global variables *Mconf*.

POSITIVELY	100
VERY – PROBABLY	75
PROBABLY	50
POTENTIALLY	25
POSSIBLY	0
POTENTIALLY – NOT	−25
PROBABLY – NOT	−50
VERY – PROBABLY – NOT	−75
NOT	−100

Table 4.1: Confidence Level Conversions

When the translator has values for the five elements of the rule, default or otherwise, it puts these elements together on a list. It then forms a larger list with an IS-AND relator at the beginning followed by the list just created and any other lists that may have been created because of adjectives or adverbs. All of these together are a summary of the information contained in a single simple statement. This list is added to a larger list that is composed of all the simple-statement lists in the premise.

Simple statements in the conclusion may be handled slightly differently. A declarative statement is treated the same as a statement in the premise. **Subject, Predicate,** and **Ending** are called and the appropriate global variables are set. If the sentence has a copula verb but no subject, only the subroutines **Predicate** and **Ending** are called, and the program assumes that the subject of the sentence is the same as the last sentence.

For an imperative sentence the translator calls **Subject** and **Find-Action**. The latter function is used exclusively for action verbs in the conclusion of the rule. Because most sentences in the conclusion of the rules are concerned with setting a variable to a certain value, the action verbs that get the most use are "assign," "increase," and "decrease." These all have the effect of changing the value of a variable, rather than setting it. The function IS-SAME, however, should be able to both set and change values. Rather than using the action verbs as the relator of the clause, the translator instead converts them into a value. "Increase" is changed to ASSIGN-HIGHER-VALUE and "decrease" to ASSIGN-LOWER-VALUE. If "assign" is used, the translator looks for an adjective that indicates level, such as "higher" or "lower." If one is found the expression translates to the same as "increase" or "decrease." If no adjective is found, the value is set to ASSIGN-VALUE.

All of the conclusion pentads are strung together in the same way as those of the premise, and the premise and conclusion are joined to form a single list. This list is returned to the calling program.

In addition to the function that converts English rule to LISP pentads, there is also a function that will convert the LISP pentads back into English. **Print-Rule** forces the different elements of the pentads into an English template. The function makes no attempt to rejoin the modifiers with the words modified, printing them instead as separate statements. Each statement is prefixed by a number; statements originating from the same simple statement have the same whole number prefix.

5 Possible Improvements

Many improvements to the Circe system are possible. As is the case with any natural language system the parser could be expanded to allow more varied sentence structures or parts of speech. It may be useful, for example, to be able to identify possessive pronouns or compound subjects. To be really useful the parser should have a built-in backup system that would allow it to try to correct errors it may have made. At present the parser has a very primitive backup system that allows it to return to the point before it started an optional branch. A better backup system would allow the program to return to the last position in the parse where a different decision could have been make and restart the parse from there. For example, if the parser found itself with an error it might backup to an exclusive-or test where a check for the first choice had been successful. It could cancel out the part of the tree that had been created for that choice and restart the parse by trying the second choice.

An interactive error correction mechanism would be another useful addition to the parser. If the parser choked on a word it could ask the user if the word was indeed the word he wanted. If it was, the parser could ask the user to specify how he wanted to use the word (which part of speech, and any other information that might be useful). A spelling corrector would also help in identifying unknown words, suggesting to the user words that have similar spellings. The system could be further sophisticated by suggesting only those words that are grammatically correct in the context. The most vital, useful addition to the parser would be a facility that allows the user to change the vocabulary interactively during parsing rather than editing the vocabulary as a separate task and then rerunning Circe on the parse that failed.

The translator could be improved by making it process simple statements. Although the parser can handle statements as well as rules, the translator is set up exclusively for rules. If this change were made the translator could be used for entering simple facts into the knowledge base, or for accepting information from a sonar operator. The program would be more useful if it were more modular. This could be done by making more use of the flavour facility of Zetalisp. Perhaps each simple statement given to the translator could be classified according to the different instances of a flavour and processed according to instructions contained within the instance. In any case, the more modular the translator is, the better, because modularity ensures that the program is more general purpose and certainly more flexible.

The system as a whole would be more useful if a rule editing facility were added. The optimum is a function that would allow the user to edit rules in an existing rule base. At the very least the user should be able to edit a rule he has just entered without completely retyping the rule. It would not be too difficult to arrange this, given that the LISP-to-English conversion function provides numbers that correspond to the position of each statement in the LISP list. For example, the statement numbered 3.2 in the English print-out is the second element of the third list in the LISP list. This information could be used to extract and edit the appropriate list from the LISP expression.

6 Design and Implementation Considerations

Many of the decisions made in the design and implementation of Circe were based on the performance of Calliope in different areas. Many of the better features of the earlier program were included in Circe, changed if necessary and improved if possible. Time and limited resources did not permit this in every case however. For example, Calliope was superior to Circe in its ability to handle vocabulary. Rather than having the grammar itself contain long lists of nouns or other words already contained in Intersensor, Calliope used Athena to consult Intersensor's knowledge base. This meant that Calliope could understand all the words contained in Intersensor, with no changes necessary if the vocabulary of Intersensor expanded or contracted. Calliope was also able to take advantage of Athena's system for handling synonyms and abbreviations.

As was noted in the introduction, the grammar of Calliope had to be severely limited in order to translate rules. This meant that only a small subset of very simple rules could be successfully parsed, ennsuring that the input to the translator was extremely predictable. Translation in this case was little more than taking key words out of one template and inserting them into another. Because Circe's grammar is comparable to that of the first Calliope, Circe's translator had to be more complex to be able to handle less predictable input. The implementation of the grammar was also changed, as a consequence of the Computer Aided Detection group of DREA acquiring Symbolics Lisp Workstations. The grammar was changed to take advantage of the flavour facility in Zetalisp, the dialect of LISP that is supported by Symbolics. Instead of using Athena's units to represent a node in the grammar tree, Circe uses instances of a flavour. A flavour is basically a user defined data structure. Each occurance of this data structure is called an instance. Both Calliope and Circe were based on an object-oriented approach to programming rather than the conventional function-oriented approach. This meant that the course of the parse was determined by information contained inside the data structures, whether units or instances, rather than a clear sequence of actions in a function.

While both programs were object oriented, Circe was written with a view towards modularity. It was decided that a system inside an environment as susceptible to change as Intersensor should be as adaptable as possible. To achieve this aim the functions in Circe are as general and as independant as possible. One of the problems with Calliope is that its tests were too rigidly defined. If a part of speech had two mandatory components followed by an optional one, there was a test written to check for that sequence. If the part of speech had one optional test followed by two mandatory ones, there was another test for that sequence. For each different combination of parts there was a different test. The parser in Circe only has tests for individual items; if a part has several components, several tests are called. This eliminates the need for defining a new test each time a new combination of parts is needed: the

existing tests are simply specified in a list in whatever order is necessary.

In keeping with modularity, the grammar is completely separate from the parser. Either could be replaced and leave the performance of the other unaffected. The vocabulary is also separate from the grammar, so that different word lists can be used for different applications.

The possibility of using a transformational grammar for Circe, rather than the simple context-free grammar now in place, was investigated. Some research had been carried out at DREA in this area [1] , [2] and some software had been written. The deep-structure representation that results from a transformational parser certainly has its advantages, especially since one of the goals of Circe was uniform representation of information. A transformational parser will give the same parse for two sentences that contain the same information, not matter how that information is organized inside the sentence. Despite the obvious benefits, it was decided that limited scope of input allowable by Circe (rules and simple statements) did not warrant the extra effort necessary to implement a transformational parser. Since uniform representation was still desired, some transformations are performed by the translator. Whether the word destined to be the object of the sentence occurs as a possessive noun or inside a prepositional phrase does not matter: the translator will understand what is meant and both will be represented the same way. Some transormations are also performed on active verbs and modifiers to ensure that the final output of the program is uniform.

The translator was originally intended to be as modular as the parser. A flavour named *Sentence-Type* was defined, and instances of this each described a possible sentence structure according to its identifying characteristics. The problem of translating, however, is not as amenable to the concept of object-centred programming as is parsing. The translator does depend for the most part on a few main functions which are relatively general purpose, but it is impossible to completely separate the functions from the input when the problem is so dependant on the input. It could be argued that a parser is just as dependant on the input, but a parser merely tells the user what exists in the sentence, rather than trying to do anything with this information. In any case, this particular translator is probably not as modular as it could be, and any change in the input conditions will necessitate some change, large or small, in the system.

7 Conclusion

Circe, like its predecessor Calliope, attempts to convert English rules into LISP-like data structures. Both programs were designed as development aids for the Intersensor knowledge based inference system. Circe performs the conversion of the rules in two major steps: it breaks the sentence down into its grammatical components and then using these components it assembles the relevant information into five-element lists. These processes work on one rule at a time, entered either by the user via the keyboard, or from a file.

The system is by no means complete or comprehensive. A parser does not exist that can understand every rule input, and this system is not leading the way. In realistic terms, the most severe limitation of the parser is its lack of a correction mechanism for decisions. The grammar can not be expanded much further without either a back-up or a look ahead facility. The translator is not without its problems either. It does not have the ability to translate single simple statements, and cannot even translate correctly all of the rules the parser can supply. This is not to say that the system is without merit; it processes a reasonable variety of rules quite successfully.

The question could legitimately be raised as to why a natural language understanding system should be used to solve this problem. No knowledge of the LISP language is needed to form the output, as the rules are not written in LISP code anyway. The output is merely a data structure that resembles LISP through its use of lists and parentheses. Presumably a sonar expert could be taught quite quickly how to organize the information. What then is the use of this program?

One of the main advantages of using the system is the consistency of representation Circe affords. If judgements are required as to how the knowledge should best be represented, as indeed they are, it is best that these judgements be consistent. Unless the same sonar operator enters every rule for the system, the judgement may not be consistent, and it may not even be consistant in that instance. The Circe system also ensures that the terminology used is consistent; any word not in the vocabulary is not understood. The same holds true for the function names: Circe acts as a filter for unwanted input. Another advantage of Circe is that it tries to store somehow every piece of information entered. Every modifier is represented in the final output. The system also gives the user a chance to see his rule rephrased in a way that shows exactly what the rule means (assuming the rule is processed correctly). Even without the improvements suggested in the previous chapter, Circe will be a useful addition to Intersensor.

Bibliography

[1] Bonner, Anthony J. *A System for Parsing Natural Language Sentences Using a Chomsky Transformational Grammar*, DREA Research Note DREA/SP/79/4, November 1979. "Informal Communication."

[2] Bonner, Anthony J. *Software Support for Transformational Parsers.* DREA Research Note DREA/SP/82/2, February, 1982. "Informal Communication."

[3] Buchanan, Bruce G., and Edward H. Shortliffe, *Rule-based Expert Systems*, Reading, Massachusetts: Addison-Wesley, 1984.

[4] Chomsky, Noam, *Syntactic Structures*, The Hague: Mouton, 1957.

[5] Dent, C. Ann, and Reid G. Smith *A Guide to Athena: A Knowledge Representation Language*, DREA Technical Memorandum 83/6, Defence Research Establishment Atlantic, Dartmouth, Nova Scotia, October 1983.

[6] Dent, C. Ann, *Reflections upon Building a Sonar Expert System*, DREA Research Note DREA/SP/85/2, January 1985. "Informal Communication."

[7] Muise, J. L., *Calliope: A Natural Language Rule Interpreter*, DREA Research Note DREA/SP/84/2, April 1984. "Informal Communication."

[8] Muise, J. L., *Calliope II: The Sequel*, DREA Research Note DREA/SP/85/3, May 1985. "Informal Communication."

[9] Rodger, R. M., *A Lisp Parser for Context-Free Grammars*, DREA Research Note DREA/SP/79/1, June 1979. "Informal Communication."

[10] *Reference Guide to Symbolics-Lisp*, Cambridge, Massachusetts: Symbolics, Inc., 1985.

[11] Shaw, Harry, *Handbook of English*, Toronto: McGraw-Hill Ryerson, 1979.

[12] Thomas, Owen, and Eugene R. Kintgen *Transformational Grammar and the Teacher of English.* New York: Holt, Reinhart and Winston, 1974.

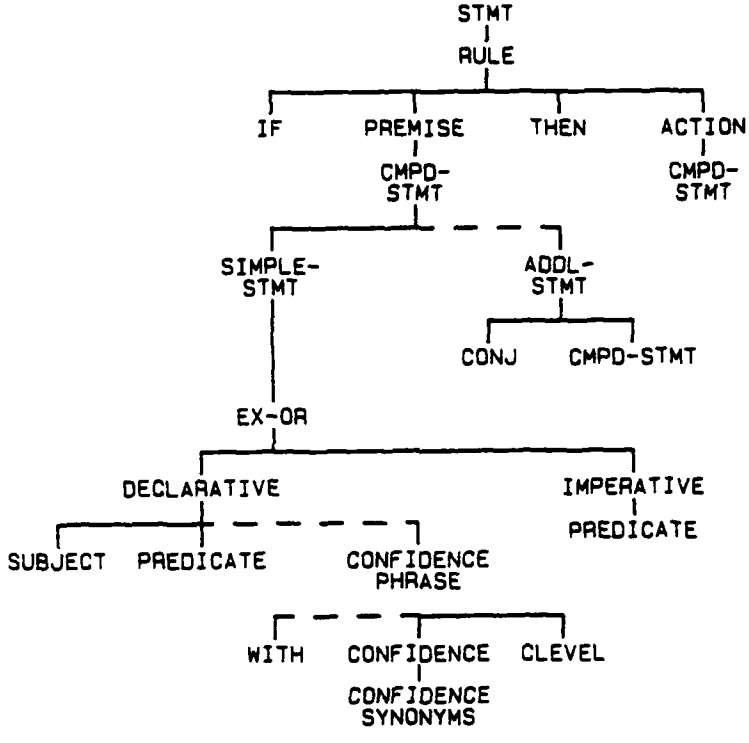
Appendix A The Grammar Expressed as BNF Rules

Stmt	$\Longrightarrow$	(/ *Rule Simple – Stmt*)
Simple – Stmt	$\Longrightarrow$	(/ *Declarative Imperative*)
Rule	$\Longrightarrow$	(& {IF} *Premise* {THEN} *Action*)
Declarative	$\Longrightarrow$	(& *Subject Predicate* (*Confidence – Phrase*))
Imperative	$\Longrightarrow$	(*Predicate*)
Premise	$\Longrightarrow$	(*Cmpd – Stmt*)
Action	$\Longrightarrow$	(*Cmpd – Stmt*)
Confidence – Phrase	$\Longrightarrow$	(& {WITH} *Confidence Clevel*)
Cmpd – Stmt	$\Longrightarrow$	(& *Simple – Stmt* (*Addl – Stmt*))
Addl – Stmt	$\Longrightarrow$	(& *Conj Cmpd – Stmt*)
Subject	$\Longrightarrow$	(*Noun – List*)
Predicate	$\Longrightarrow$	(/ *Active – Pred Inactive – Pred*)
Noun – List	$\Longrightarrow$	(& *Noun – Phrase* (*Addl – Noun – List*))
Noun – Phrase	$\Longrightarrow$	(& *Substantive* (*Prep – Phrase*))
Addl – Noun – List	$\Longrightarrow$	(& *Conj Noun – List*)
Substantive	$\Longrightarrow$	(& (*Article*) *Modified – Noun*)
Article	$\Longrightarrow$	(**article**)
Modified – Noun	$\Longrightarrow$	(& (*Adj – List*) (/ *Noun Pronoun*))
Adj – List	$\Longrightarrow$	(& *Adj – Phrase* (*Adj – List*))
Adj – Phrase	$\Longrightarrow$	(/ *Poss – Noun Adjective*)
Poss – Noun	$\Longrightarrow$	(**noun** +' **s**)
Adjective	$\Longrightarrow$	(**adjective**)
Noun	$\Longrightarrow$	(**noun**)
Pronoun	$\Longrightarrow$	(**pronoun**)
Prep – Phrase	$\Longrightarrow$	(& *Preposition Substantive*)
Preposition	$\Longrightarrow$	(**preposition**)

Appendix A

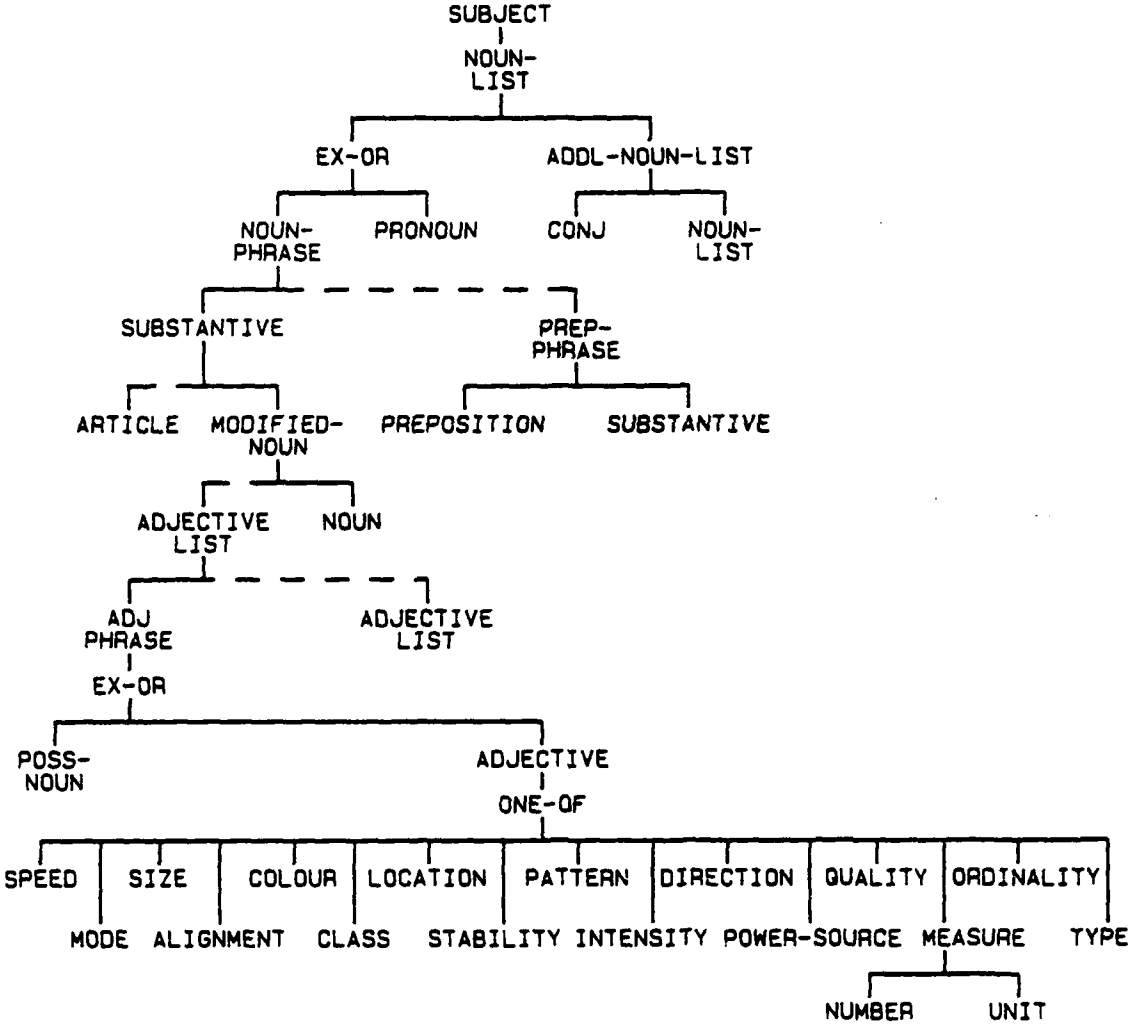
Active – Pred	$\Longrightarrow$	(& *Active – Verb – Phrase* (*Objects*))
Inactive – Pred	$\Longrightarrow$	(& *Copula – Verb* (*Copula – Adverb*) (*Confidence – Level*) *Completion*
Active – Verb – Phrase	$\Longrightarrow$	(*Active – Verb* (*Adverb – Phrase*))
Objects	$\Longrightarrow$	(& (*Direct – Object*) (*Indirect – Object*))
Active – Verb	$\Longrightarrow$	(**active – verb**)
Adverb – Phrase	$\Longrightarrow$	(& *Adverb* (*Adverb – Phrase*))
Adverb	$\Longrightarrow$	(**adverb**)
Direct – Object	$\Longrightarrow$	(& *Noun – Phrase* (*Complement*))
Complement	$\Longrightarrow$	(*Adj – Complement Noun – Complement*)
Adj – Complement	$\Longrightarrow$	(*Adj – Phrase*)
Noun – Complement	$\Longrightarrow$	(*Substantive*)
Indirect – Object	$\Longrightarrow$	(& *Preposition Noun – Phrase* (*Direct – Object*))
Copula – Verb	$\Longrightarrow$	(**copula – verb**)
Copula – Adverb	$\Longrightarrow$	(**copula – adverb**)
Confidence – Level	$\Longrightarrow$	(**confidence – level – adverb**)
Completion	$\Longrightarrow$	(/ *Pred – Adj Pred – Noun*)
Pred – Adj	$\Longrightarrow$	(*Adj – List*)
Pred – Noun	$\Longrightarrow$	(*Objects*)

Appendix B The Grammar Displayed as a Tree

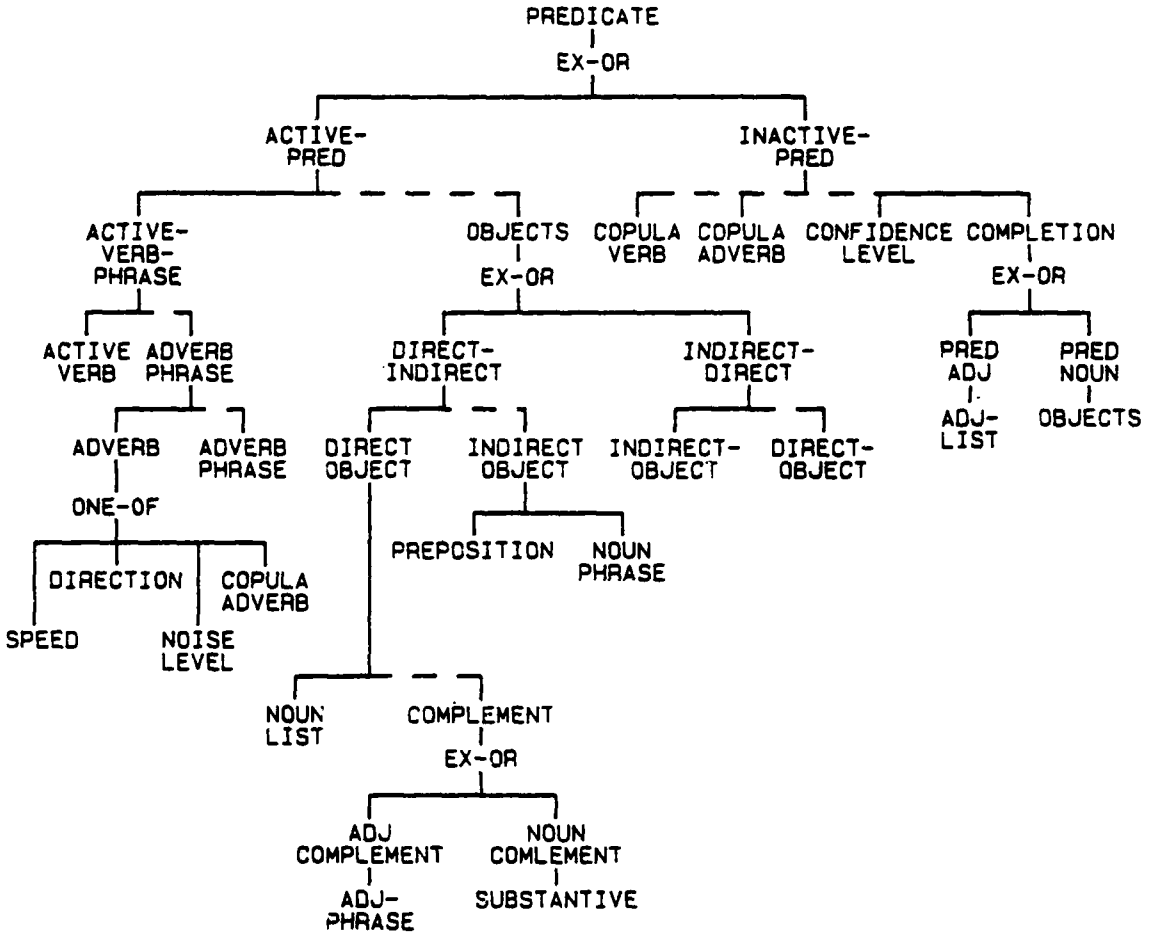


A solid line indicates that the node is mandatory, while a broken one means the part is optional.

Appendix B


SUBJECT
NOUN-LIST
EX-OR
ADDL-NOUN-LIST
NOUN-PHRASE
PRONOUN
CONJ
NOUN-LIST
SUBSTANTIVE
PREP-PHRASE
ARTICLE
MODIFIED-NOUN
PREPOSITION
SUBSTANTIVE
ADJECTIVE LIST
NOUN
ADJ PHRASE
ADJECTIVE LIST
EX-OR
POSS-NOUN
ADJECTIVE
ONE-OF
SPEED
SIZE
COLOUR
LOCATION
PATTERN
DIRECTION
QUALITY
ORDINALITY
MODE
ALIGNMENT
CLASS
STABILITY
INTENSITY
POWER-SOURCE
MEASURE
TYPE
NUMBER
UNIT

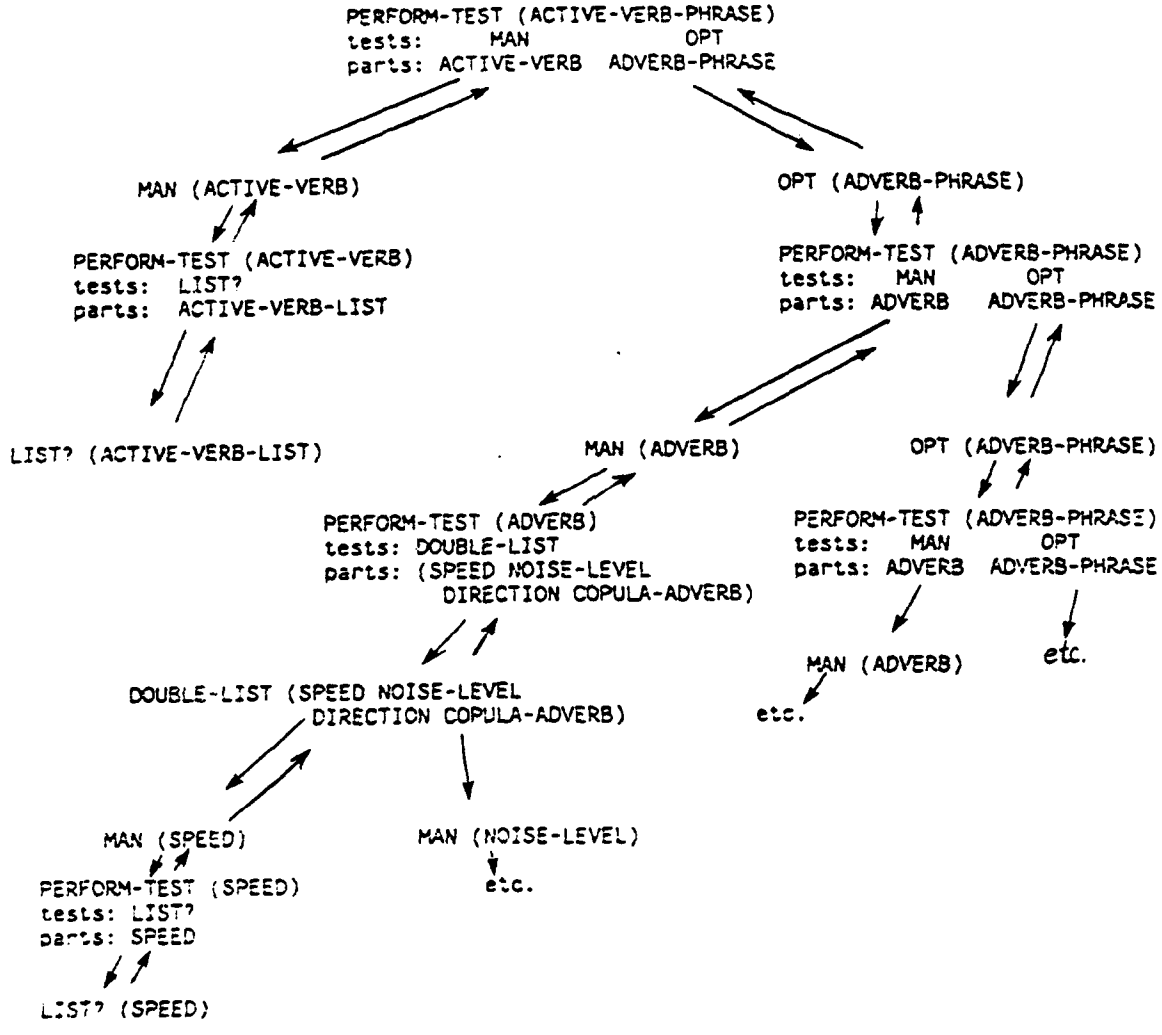
Appendix B


PREDICATE
EX-OR
ACTIVE-PRED
INACTIVE-PRED
ACTIVE-VERB-PHRASE
OBJECTS
EX-OR
COPULA VERB
COPULA ADVERB
CONFIDENCE LEVEL
COMPLETION
EX-OR
ACTIVE VERB
ADVERB PHRASE
DIRECT-INDIRECT
INDIRECT-DIRECT
PRED ADJ
PRED NOUN
ADJ-LIST
OBJECTS
ADVERB
ADVERB PHRASE
DIRECT OBJECT
INDIRECT OBJECT
INDIRECT-OBJECT
DIRECT-OBJECT
ONE-OF
PREPOSITION
NOUN PHRASE
DIRECTION
COPULA ADVERB
SPEED
NOISE LEVEL
NOUN LIST
COMPLEMENT
EX-OR
ADJ COMPLEMENT
NOUN COMLEMENT
ADJ-PHRASE
SUBSTANTIVE

Appendix C Examples of the Grammar Implemented in Circe

```
(defflavor sentence-parts (test parts significant?) ()
      :gettable-instance-variables
      :initable-instance-variables
      :settable-instance-variables)

(defvar stmt nil)

(setq stmt (make-instance 'sentence-parts :test '(exor)
                          :parts '((rule simple-stmt))
                          :significant? 'nil))
(defvar rule nil)

(setq rule (make-instance 'sentence-parts :test '(man man man man)
                          :parts '(if premise then action)
                          :significant? nil))

(defvar simple-stmt nil)

(setq simple-stmt (make-instance 'sentence-parts :test '(exor )
                                 :parts '((declarative imperative))
                                 :significant? T))

(defvar declarative nil)

(setq declarative (make-instance 'sentence-parts
                   :test '(man man opt)
                   :parts '(subject predicate confidence-phrase)
                   :significant? T))

(defvar imperative nil)

(setq imperative (make-instance 'sentence-parts :test '(man)
                                :parts '(predicate )
                                :significant? T))

(defvar confidence-phrase nil)

(setq confidence-phrase (make-instance 'sentence-parts
                                :test '(opt man man)
                                :parts '(with confidence clevel)
                                :significant? t))
```

Appendix D An Example of How the Parser Works



The arrows indicate either calls to or returns from functions or, in the case of Perform-test, a message being sent to an instance.

Appendix E Sample Output

```
;;; The following are rules that were generated by Circe and written into a file
;;; The rules have three parts to them:  a rule name, the original English input,
;;; and the lisp rule generated by the translator.


(TEST-RULE1
 IF LOFAR-FEATURE IS WEAK WITH CONFIDENCE-LEVEL 80
 THEN IT IS FROM A DISTANT TARGET.
((IS-AND (IS-AND (IS-SAME LOFAR-FEATURE INTENSITY WEAK 80)))
 (IS-AND (CONCLUDE (IS-SAME LOFAR-FEATURE ATTRIBUTE TARGET 100)
                   (IS-SAME TARGET QUALITY DISTANT 100)))))

(TEST-RULE2
IF THE FREQUENCY-SPECTRUM OF THE SECOND HARMONIC-SET IS WIDE
THEN INCREASE THE SMALLEST INTERVAL OF THE LOFAR-DISPLAY
((IS-AND (IS-AND (IS-SAME HARMONIC-SET FREQUENCY-SPECTRUM WIDE 100)
                 (IS-SAME HARMONIC-SET ORDINALITY SECOND 100)))
(IS-AND (CONCLUDE (IS-SAME LOFAR-DISPLAY INTERVAL ASSIGN-HIGHER-VALUE 100)
                  (IS-SAME INTERVAL SIZE SMALLEST 100)))))

(TEST-RULE3
IF THE PROPELLER'S BLADE ROTATES QUIETLY
THEN DETECTION OF CAVITATION IS VERY-PROBABLY SMALL
((IS-AND (IS-AND (ROTATES PROPELLER BLADE QUIETLY 100)))
(IS-AND (CONCLUDE (IS-SAME CAVITATION DETECTION SMALL 75)))))

(TEST-RULE4
IF MANY TARGET COMPETE FOR IDENTIFICATION
THEN ASSIGN THE HIGHEST PRIORITY TO THE FASTEST TARGET.
((IS-AND (IS-AND (COMPETE TARGET ATTRIBUTE IDENTIFICATION 100)
                 (IS-SAME TARGET CARDINALITY MANY 100)))
(IS-AND (CONCLUDE (IS-SAME TARGET PRIORITY ASSIGN-VALUE 100)
                  (IS-SAME PRIORITY QUALITY HIGHEST 100)
                  (IS-SAME TARGET SPEED FASTEST 100)))))

;;; The results of test-rule4 may seem a bit confusing.  If you examine the
;;; list closely, however, you will find that the rule does indeed reflect the
;;; information in the english input
```

Appendix E

```
(TEST-RULE5
IF PLATFORM HAS A STABLE AUXILIARY AND HAS A SINGING PROPELLER
 THEN IT IS VERY-PROBABLY A SMALL-FISHING-TRAWLER VESSEL
((IS-AND (IS-AND (HAS PLATFORM ATTRIBUTE AUXILIARY 100)
                 (IS-SAME AUXILIARY STABILITY STABLE 100))
         (IS-AND (HAS ATTRIBUTE ATTRIBUTE PROPELLER 100)
                 (IS-SAME PROPELLER TYPE SINGING 100)))
 (IS-AND (CONCLUDE (IS-SAME PLATFORM ATTRIBUTE VESSEL 75)
                 (IS-SAME VESSEL CLASS SMALL-FISHING-TRAWLER 100)))))

;;; The results of test-rule5 are confusing and perhaps not what was meant
;;; by the english input. The confusion results from the small amount of information
;;; input.  The translator has a hard time filling the gaps in information, and as
;;; a result the second statement in the premise clause is not represented well.

(TEST-RULE6
 IF A LOFAR-DISPLAY HAS A FIGURE-EIGHT PATTERN
UNDER INTERFERENCE-PATTERN
THEN THE SOURCE OF IT IS THE SMALL RED OBJECT
BESIDE THE LARGE-COMMERCIAL-SURFACE VESSEL.
 ((IS-AND (IS-AND (HAS LOFAR-DISPLAY ATTRIBUTE PATTERN 100)
                  (IS-SAME PATTERN PATTERN FIGURE-EIGHT 100)))
 (IS-AND (CONCLUDE (IS-SAME TARGET SOURCE OBJECT 100)
                  (IS-SAME OBJECT SIZE SMALL 100)
                  (IS-SAME OBJECT COLOUR RED 100)))))

;;; Test-rule6 is the opposite of  test-rule5.  Here too much information
;;; was supplied in each simple statement.  If the user had supplied the rule:
;;; "If the lofar-display has a figure-eight pattern and the pattern is under
;;; interference-pattern then the source of it is the small red object and the
;;; object is beside the large-commercial-surface vessel." the translator would
;;; had performed better.  Admittedly this is a bit stilted and may not seem
;;; to say the same thing, but it does produce the desired results.

(TEST-RULE7
IF THE FIRST TARGET'S DIESEL ENGINE IS GREATER-THAN 5 TONS
AND THE 100 FOOT LONG SHAFT OF IT IS VERY-PROBABLY TWIN-SHAFTED
THEN THE TARGET IS A NEUTRAL SURFACE-VESSEL WITH CONFIDENCE 60
AND DECREASE THE PRIORITY OF THE US-NAVAL-SURFACE HYPOTHESIS.
((IS-AND (IS-AND (IS-GREATER-THAN TARGET ENGINE STONS 100)
                 (IS-SAME TARGET ORDINALITY FIRST 100)
                 (IS-SAME ENGINE POWER-SOURCE DIESEL 100))
         (IS-AND (IS-SAME TARGET SHAFT TWIN-SHAFTED 75)
                 (IS-SAME SHAFT LENGTH 100FT 100)))
(IS-AND (CONCLUDE (IS-SAME TARGET ATTRIBUTE SURFACE-VESSEL 60)
                 (IS-SAME SURFACE-VESSEL ALIGNMENT NEUTRAL 100))
        (CONCLUDE (IS-SAME HYPOTHESIS PRIORITY ASSIGN-LOWER-VALUE 100)
                  (IS-SAME HYPOTHESIS CLASS US-NAVAL-SURFACE 100)))))
```

Appendix F How to make Changes to the System

F.1 Grammar Changes

Changing the grammar of the program is a simple matter, much the same as adding or deleting a node from a linked list. To add a new node to the grammar tree, the name and a test to be used for it must be inserted into the node directly above it. This is achieved by adding the name of the new node to its ancestors's part-list, and adding the test to the corresponding position in the test-list. After this the new node can be defined by making it an instance of the flavour *Sentence-parts*. At the same time the component parts of the node and their associated testscan be specified, along with another variable, Significant?. This logical variable specifies whether the node-name should be placed in the parse-list if the node is found.

Suppose, for example, you wanted to change the grammar so that a confidence phrase could be used in an imperative simple-statement. All that would be necessary here is an addition to the initialization commmand that creates the instance IMPERATIVE. **Opt** would be placed at the end of the test list, and CONFIDENCE-PHRASE at the end of the parts list. If you wanted to create a different type of confidence phrase, perhaps allowing a relator such as "greater-than," you would follow the same procedure, adding the name of the new confidence phrase to the parts list of IMPERATIVE. After that you would have to make a new instance of *Sentence-parts* using the name of the new confidence phrase. The commands necessary would look something like this (depending on which parts and which test were desired):

```
(defvar NEW-CONFIDENCE-PHRASE nil)

(setq NEW-CONFIDENCE-PHRASE (make-instance 'SENTENCE-PARTS
:test '(OPT MAN OPT MAN)
:parts '(WITH CONFIDENCE-SYNONYMS RELATOR CLEVEL)
:significant? T))
```

The user must be careful when specifying certain tests. **Exor**, **OOB** (one-or-both), and **Double-List** all take more than one part. The parts associated with these then must be enclosed in parentheses, inside the parts list.

If the new node is to be inserted between an ancestor and its descendant, the name of the descendant should be put on the parts list of the new node, and its test on the test list.

To delete a node from a tree it is a simple matter of removing its name from the parts list of its ancestor, and the test from the test list. Again, if the node to be deleted has descendants, and the user wishes to keep them in the grammar tree, their names and tests should be inserted

into the ancestor node.

F.2 Adding New Tests

If the user finds that he needs a new test for a certain node, he need only define the function and insert its name into the test-list where the node is called. The program will be clearer and less cluttered if the user defines his function using existing tests. Both **Exor** and **OOB**, for example, were written by combining the basic functions **man** and **Opt** using different conditions. With a basic understanding of how the program works and a bit of imagination the user should be able to write almost any test using only the existing ones. If the user writes a function that checks for a terminal node, the function should include a call to the function **Gettoken** to get a new word. The function should also set the variable *word?* which will ensure that the word itself, not just the type of the word, is added to the parse list.

F.3 Adding to the Vocabulary

Adding to the vocabulary is an easy if inconvenient job. New words are simply inserted into the appropriate list at the beginning of the file containing the parser. Note that adding the word "fill" to the noun list does not mean that the program will understand "fill" if it is used as a verb. If the word being added to the vocabulary normally contains a single slash, the word should be input with a double one.

Appendix G A Listing of the Parser's Functions

```
;;; perform-test is the driving routine of the parser.  It gets  from  the sentence component
;;; that called it the different parts that must be present for that sentence component to exist.
;;; It tests for the presence of these by calling the test specified for each of the component parts.

(defmethod (sentence-parts :perform-test) ()
  (prog (test-list parts-list ind-test ind-part
         success result-list answer )
        (setq test-list test)
        (setq parts-list parts)
        (setq result-list ())
        (setq answer ())
        loop
        (setq ind-test (car test-list))
        (setq ind-part (car parts-list))
        (setq test-list (cdr test-list))
        (setq parts-list (cdr parts-list))
        (cond
          ((setq success (funcall ind-test ind-part))
           (cond
             ((eq ind-part 'simple-stmt)
              (setq stmtnum (add1 stmtnum))
              (setq success (list success))))
           (cond
             (result-list
                (setq result-list (append result-list success)))
             (T (setq result-list  success))))
          ((equal ind-test 'opt)
           (setq success T)))
        (cond ((not success)
               (return nil))
              ((null test-list)
                (cond
                  (answer
                    (setq answer (append result-list answer )))
                  (T (setq answer result-list)))
               (return  answer))
              (T (go loop)))))
```

Appendix G

```
;;; Man tests for the existence of one sentence-part by calling perform-test.  If the
;;; part exists, it returns the part; if not, it returns nil, as the part is mandatory.

(defun man (part)
  (prog (result signif)
      (cond
        ((setq result (send (eval part) ':perform-test))
         (cond (word?
                (setq result (list (cons part result)))
                (setq word? nil))
               ((setq signif (send (eval part) ':significant?))
                (setq result (append (list (cons part stmtnum)) result)))
               (T
                (setq result (append (list (cons part 'd)) result))))
         (setq error nil)
         (and trc2 (format T " is a ~a ~%" part))
         (return result))
        (T
         (cond
           (part-error)
           (T
            (setq part-error part)))
         (and trc2 (format T " no ~a ~%" part ))
         (return nil)))))


;;; Opt checks for  the existence of an optional sentence-part.  It does this by calling
;;; perform-test for the sentence-part it was supplied.


(defun opt (part)
  (prog (result signif temp-start temp-end temp-word)
        (setq temp-start start-word)
        (setq temp-end end-word)
        (setq temp-word word)
        (cond
          ((setq result (send (eval part) ':perform-test))
           (cond (word?
                  (setq result (list (cons part result)))
                  (setq word? nil))
                 ((setq signif (send (eval part) ':significant?))
                  (setq result (append (list  (cons part stmtnum)) result)))
                 (T
                  (setq result (append (list (cons part 'd)) result))))
           (setq error nil)
           (and trc2 (format T " forms an opt ~a ~% " part))
           (return result))
          (T (and trc2 (format T " no opt ~a ~% " part))
             (setq error nil)
             (setq start-word temp-start)
             (setq end-word temp-end)
             (setq word temp-word)
             (return nil)))))

;;; Exor is supplied two sentence-parts, only one of which should exist. The first sentence-part
;;; is checked, and if it is found to exist, the search stops.  If it does not exist the second part
;;; is checked for.  If this does not exist either the error flag is set.

(defun exor (part)
  (prog (result)
        (cond
          ((setq result (man (car part)))
           (return result))
          ((setq result (man (cadr part)))
           (return result))
          (T (return nil)))))
```

Appendix G

```
;;; Oob is given two sentence-parts, one or both of which can exist, but only in the order supplied.

(defun oob (part)
  (prog (result result2)
        (cond
          ((setq result (man (car part)))
           (cond
             ((setq result2 (man (cadr part)))
              (return (list result result2)))
             (T (return result))))
          ((setq result (man (cadr part)))
           (return result))
          (T (return nil)))))

;;; Double-list checks a list of parts to find the first occurance of one of the parts.

(defun double-list (list)
  (prog (signif slist result)
        loop
        (setq slist (car list))
        (setq list (cdr list))
        (cond
          ((setq result (man slist))
           (cond
             ((setq signif (send (eval slist) ':significant?))
              (setq result (append (list (cons slist stmtnum)) result)))
             (T
              (setq result (append (list (cons slist 'd)) result))))
           (return result))
          (T
           (cond
             (list
              (go loop))
             (T
              (return nil)))))))

;;; List? checks whether the word currently being examined is on the word-list it has been supplied.

(defun list? (wordlist)
  (prog (temp-word)
        (cond
          ((null word) nil)
          (T (cond
               ((memq word (eval wordlist))
                (format T " ~a " word)
                (setq temp-word word)
                (setq word (gettoken))
                (setq word? T)
                (return temp-word))
               (T (setq error word)
                  (return nil)))))))

;;; Spec? tests the word currently being processes against a specific word supplied to it.

(defun spec? (word1)
    (prog ()
          (cond
            ((null word) nil)

            (T (cond
                 ((equal word word1)
                  (format T " ~a " word)
                  (setq word (gettoken))
                  (setq word? T)
                  (return word1))
                 (T (setq error word)
                    (return nil)))))))
```

```
;;;Number? simply checks whether or not the word currently being examined is a number

(defun number? (&rest ignore)
  (prog (number)
        (cond
          ((numberp word)
           (format T " -d " word)
           (setq number word)
           (setq word (gettoken))
           (setq word? T)
           (return number))
          (T
           (return nil)))))

;;; Main is the routine which controls the processing of the sentence.  It prompts the user
;;; for the sentence to be parsed and then calls perform-test.  If the sentence is successfully
;;; parsed, the parse tree is output along with other information and the user is prompted
;;; to enter another sentence.

(defun CIRCE ()
  (prog (eof no-commas temp stopflg lrule rule-name sentdescr temp-user-in )
        (setq stopflg nil)
        (setq session-list nil)
        (setq temp-user-in nil)
        (setq eof nil)
        (setq user-in T)
        (setq loaded-file nil)

        (choose-user-options *choose-variables-list* ':function 'readit)



;;;     (setq trc1 (y-or-n-p " Do you want to see the results of the parse? "))
;;;     (setq trc2 (y-or-n-p " Tracing messages? "))
        (loop while (not stopflg) do
          (progn
            (cond
              ((or user-in temp-user-in)
               (setq temp-user-in nil)
               (setq paragraph (string-upcase (with-input-editing-options
                                                ((:input-history-default sentdescr))
                                              (prompt-and-read :string
                  " Enter the rules in complete sentences.  To stop, type the word STOP. To
change user-options type OPTIONS.
")))))
              (T
                  (cond
                    ((y-or-n-p " Process rule from file? ")
                     (setq paragraph (car input-list))
                     (setq input-list (cdr input-list))
                     (cond
                       ((equal ""  paragraph)
                        (setq stopflg T)
                        (format t "~% The current file is empty.")
                        (COND
                          ((Y-OR-N-P " Quit (y) or change options (n)? ")
                           (SETQ PARAGRAPH "STOP"))
                          (t
                           (setq paragraph "OPTIONS"))))))
                    (T
                     (cond
                       ((y-or-n-p " Quit (y) or change options (n)? ")

                        (setq paragraph "STOP"))
                       (T
                        (setq paragraph "OPTIONS")))))))))
```

```
            (setq sentdescr (string paragraph))
            (cond
              ((equal (string-right-trim '(#\.) paragraph) "options")
               (choose-user-options *choose-variables-list* ':function 'readit))
              ((not (equal (string-right-trim '(#\.) paragraph) "STOP"))
               (setq no-commas nil)
               (setq start-word 0)
               (setq end-word 0)
               (setq stmtnum 1)
               (setq eos nil)

               (loop while (not no-commas) do          ;;; This loop turns all commas into blanks
                 (progn
                   (setq temp (string-search-char <COMMA> paragraph))
                   (cond
                     ((null temp) (setq no-commas T))
                     (T (aset <BLANK> paragraph temp)))))

               (setq error nil)
               (setq word (gettoken))
               (setq fanswer (send stmt ':perform-test))
               (cond
                 (error (format T "~% Error: Looking for a ~a but found ~a instead ~%"
                                part-error error)
                        (setq paragraph ""))
                 ((not eos)
                  (format T "ERROR: Finished before end of sentence."))
                 (T
                  (cond
                    (trc1
                     (grind-top-level fanswer)))
                  (setq lrule (make-rule fanswer))
                  (grind-top-level lrule)
                  (print-rule lrule)
                  (cond
                    ((y-or-n-p " Is this what you meant? ")
                     (setq rule-name (prompt-and-read ':expression "Enter a name for the rule "))
                     (setq lrule (list rule-name sentdescr lrule))
                     (setq session-list (append session-list (list lrule))))
                    (T
                     (format T
                       "~%  Re-enter the rule, rephrasing the incorrectly parsed portion")
                     (setq temp-user-in t))))))

              (T
               (setq stopflg T)))))
            (write-to-file session-list)
            (return T)))

;;; Gettoken extracts the next word from the sentence input by the user.  It changes
;;; the word into uppercase letters and converts it from a character string to an atom
;;; to make it easier to process.  Any blanks or punctuation marks are stripped from
;;; the word, and apostrophes are converted to number signs (#).


(defun gettoken ()
  (prog (word temp length location)
        (cond
          (eos (return nil))
          ((not (null end-word))
           (setq start-word (string-search-not-char <BLANK> paragraph end-word))
           (setq end-word (string-search-char <BLANK> paragraph start-word))
           (setq word (substring paragraph start-word end-word))
```

```
        (setq word (string-right-trim '(#\.) word))
        (cond
          ((setq location (string-search-char <APOS> word)) ;;; doesn't handle more than one - fix
           (aset #\# word location)
           (setq pword word)))
        (cond
         ((multiple-value (temp length) (parse-number word 0))
          (cond
            ((equal length (string-length word))
             (return temp)))))
         (return (intern word)))
        (T (setq eos T)
           (return nil)))))

(defun pnoun? (word-list)
  (prog (word1)
        (ignore word-list)                    ; later check wordlist supplied by calling routine...
        (cond
          (pword
           (setq word1 (intern (string-right-trim "#S" pword)))
           (cond
             ((memq word1 n-list)
              (format T " ~a " word)
              (setq word (gettoken))
              (setq word? T)
              (setq pword nil)
              (return word1))
             (T
              (setq error word)
              (return nil))))
          (T
           (setq error word)
           (return nil)))))
(defun write-to-file (lispex)
  (prog (file )
        (cond
          (session-list

           (setq file (prompt-and-read '(:pathname :default "bl:>JIM>rules.lisp")
                                       "~% Enter a file name to write to (default is ~a) ~%"
                                       "bl:>jim>rules.lisp"))
           (with-open-file (str file
                                ':direction ':output
                                ':characters t
                                ':if-exists ':append
                                ':if-does-not-exist ':create)

             (dolist (element lispex)
               (send str :string-out (format nil "~a" element))))
           (format T "~% Rules written to ~A" file))
          (t
           (format t "~% No rules to write")))))

(defun readit (window variable old-value new-value)
  (prog (paragraph end?)
        (ignore window old-value new-value)
        (cond
          ((OR (eq variable 'user-in) (EQ VARIABLE 'FILE))
              (cond
                ((eq file loaded-file))
                (t
                 (setq input-list nil)
                 (with-open-file (str file
                                      ':direction ':input
                                      ':characters t)
                   (loop while (not end?) do
                     (multiple-value (paragraph end?) (send str ':line-in))
                     (setq input-list (append input-list  (list paragraph)))))
                 (setq loaded-file file)))))
        (return nil)))
```

Appendix H A Listing of the Translator

```
;;;       >>> describes the different sentence-structure types.

(defflavor sentence-types (parts ending)
           ()
      :gettable-instance-variables
      :settable-instance-variables
      :initable-instance-variables)

(defvar s1 nil)

(setq s1 (make-instance 'sentence-types :parts '(subject active-verb) :ending '()))
                                        ;;; example : "Fish swim."

(defvar s2 nil)

(setq s2 (make-instance 'sentence-types :parts '(subject active-verb direct-object)
                        :ending '((direct-object . noun))))
                                        ;;; example : "Jim writes programs."

(defvar s3 nil)

(setq s3 (make-instance 'sentence-types
                        :parts '(subject active-verb indirect-object direct-object)
                        :ending '((IN . noun))))
                        ;;; example: "He gave her presents."

(defvar s4 nil)

(setq s4 (make-instance 'sentence-types :parts '(subject copula-verb pred-noun)
                        :ending '((pred-noun . noun))))
                                        ;;; example: "They are aliens."

(defvar s5 nil)

(setq s5 (make-instance 'sentence-types :parts '(subject copula-verb pred-adj)
                        :ending '((pred-adj . adjective))))
                                        ;;; example: "She is ugly."

(defvar s6 nil)

(setq s6 (make-instance 'sentence-types
                        :parts '(subject active-verb direct-object noun-complement)
                        :ending '((direct-object . noun) (noun-complement . noun))))
                        ;;; example: "I called him a fool."

(defvar s7 nil)

(setq s7 (make-instance 'sentence-types
                        :parts '(subject predicate direct-obj adj-complement)
                        :ending '((direct-obj . noun) (adj-complement . adjective))))
                        ;;; example: "Ann painted the bicycle purple."

(defvar action-convert nil)

(setq action-convert '((assign . assign) (increase . assign-higher)
                       (decrease . assign-lower)))
```

Appendix H

```
(setq direction-check '((northerly . north) (northern . north) (southerly . south)
                        (southern . south) (easterly . east) (eastern . east)
                        (westerly . west) (western . west)))

(defvar quality-check nil)

(setq quality-check '((long . length) (away . distance) (distant . distance) (high . height)
                      (tall . height) (deep . depth) (wide . width)))

(defvar predicate-check nil)

(setq predicate-check '((greater-than-or-equal-to . not-less-than)
                        (less-than-or-equal-to . not-greater-than)))

(defvar speed-check nil)

(setq speed-check '((faster . fast) (quickly . fast) (quick . fast) (slowly . slow)
                    (slower . slow)))

(defvar unit-check nil)


(setq unit-check '((hertz . hz) (feet . ft) (foot . ft) (pounds . lbs) (meters . m)
                   (metres . m) (meter . m) (metre . m) (kilometre . km) (kilometer . km)
                   (kilometres . km) (kilometers . km) (inches . in) (knots . kts)))

(defvar unit-convert nil)

(setq unit-convert '((hz . frequency) (lbs . weight) (tons . displacement) (kts . velocity)
                     (km . distance) (in . length) (m . length) (ft . length)))


(defvar curclause nil)

(defvar default-confidence nil)

(setq default-confidence 100)

(defvar default-relation nil)

(defvar fnum nil)

(defvar frame1 nil)

(defvar mrel nil)

(defvar mobj nil)

(defvar matt nil)

(defvar mval nil)

(defvar mconf nil)

(defvar confidence-convert nil)

(setq confidence-convert '((positively . 100) (very-probably . 75) (probably . 50)
                           (potentially . 25) (possibly . 0) (potentially-not . -25)
                           (probably-not . -50) (very-probably-not . -75) (not . -100)))

(defvar direction-check nil)
```

Appendix H

```
(defvar pobj nil)

(defvar patt nil)

(defvar pval nil)

(defvar sentence-type nil)


;;; classify identifies the basic structure of the sentence (one of seven different structure types)

(defun classify (stmt)
  "Classify examines the parse of a simple statement to determine what type of structure it
has."
     (prog ()
           (cond
             ((assq 'copula-verb stmt)

              (cond
                ((assq 'pred-noun stmt)
                 (return 's4))
                (T
                 (return 's5))))
              (T
                (cond
                  ((assq 'objects stmt)
                   (cond
                     ((assq 'direct-object stmt)
                      (cond
                        ((assq 'complement stmt)
                         (cond
                           ((assq 'noun-comp stmt)
                            (return 's6))
                           (T
                            (return 's7))))
                        (T
                         (return 's2))))
                     (T
                      (return 's3))))
                  (T
                   (return 's1)))))))


;;; Breaker separates the parse tree into the component statements and returns the number of statements
in the parse

(defun breaker (Brule)
  "Breaker separates the parse tree into the component statements and returns the number of
statements in the parse."
  (prog (subnum substmt)
        (setq brule (cddr brule))
        (setq subnum 1)
        loop1
        (setq substmt (car brule))
        (setq brule (cdr brule))
        (cond
          (substmt
           (cond
             ((equal (car substmt)  (cons 'simple-stmt  subnum))
              (set (intern (format nil "SUBSTMT~D" subnum)) substmt)
              (setq subnum (add1 subnum))))
           (go loop1))
          (T
           (return subnum)))))
```

```
;;; Subject puts the subject of the sentence into the global variable "matt" and puts
;;; any prepositional noun or possessive noun into "mobj."  If there are adjectives modifying
;;; any of these, the subroutine Extract-adj is called to make a separate frame to include
;;; each adjective.

(defun subject (subphrase)
  "Subject identifies the subject of the phrase supplied to it, as well as any prepositional
nouns, possessives, or modifiers."
     (prog (sub1 subsub  temp pro)
                                                ;;; get the possessive noun first -  if there is one


     (cond
        ((setq temp (cdr (assq 'poss-noun subphrase)))
        (setq mobj temp)
        (setq subphrase (extract-adj subphrase 'poss-noun mobj))
        (SETQ POBJ MOBJ)))

                                           ;;; get the subject of the sentence (if it is a
                                           ;;; pronoun it takes the subject from the previous sentence.

     (setq sub1 (ldiff subphrase (memass 'eq 'prep-phrase subphrase)))
     (cond
       ((setq pro (assq 'pronoun sub1))
        (SETQ MATT PATT))
       ((setq temp (cdr (assq 'noun sub1)))
        (setq matt temp)
        (setq sub1 (extract-adj sub1 (car (rassq matt sub1)) matt))
        (SETQ PATT MATT)))

                                                ;;; finds the noun in the prepositional phrase,
                                                ;;; if there is one
     (cond
       ((setq subsub (memass 'eq 'prep-phrase subphrase))
        (COND
          ((setq pro (assq 'pronoun subphrase))
           (SETQ MOBJ POBJ))
          (T
           (setq mobj (find-after 'preposition 'noun subsub))
           (extract-adj subsub (car (rassq mobj subphrase)) mobj)
           (SETQ POBJ MOBJ)))))
     (return T)))

;;; Predicate finds the predicate in the statement being examined.  It first decides whether
;;; the verb in question is an active or a copula verb.  If it is a copula verb it searches for
;;; any adverbs that might qualify the verb, such as  "greater-than" or "less-than."  If the
;;; copula verb is "equals" or "is" it is converted to something more uniform. The routine also
;;; checks if there is a confidence level indicator following the copula verb, such as "possibly" or
;;; "probably-not."
;;;     If the verb is an active verb it is used as the relator.  The routine checks for adverbs
;;;  after the verb such as "quickly" or "quietly," and, if any are found, converts them to
;;; their adjective form and creates a frame for them.

(defun predicate (subphrase)
  "Predicate identifies the verb in the phrase or sentence supplied to it.  It does not
return any variables; it does however set the global variable mrel, and it may change the
global variable mconf."
  (prog (kill subsub advrb  END copverb  att val temp)
        (cond
                                                ;;; checks if the verb is copula
```

```
        ((setq copverb (find-after 'predicate 'copula-verb subphrase))
         (setq kill (memass 'eq 'objects subphrase))
         (setq subsub (ldiff subphrase kill))
         (cond
           ((setq advrb (find-after 'predicate 'copula-adverb subsub))
            (setq mrel (intern (format nil "~D-~D" copverb advrb))))
           ((eq copverb 'equals)
            (setq mrel 'is-equal-to))
           ((eq copverb 'is)
            (setq mrel 'is-same))
           (T
            (setq mrel copverb)))
         (cond
           ((setq advrb (find-after 'predicate 'confidence-level subsub))
            (setq mconf (cdr (assq advrb confidence-convert))))))

                                              ;;; The verb must be active
        (T
         (setq mrel (find-after 'predicate 'active-verb subphrase))
         (SETQ END (SEND (EVAL SENTENCE-TYPE) :ENDING))
         (cond
                                                        ;;; If the ending of the sentence is nil
                                                        ;;; do not remove the adverb if there is one
                                                        ;;; It will be used as the value of the
                                                        ;;; primary clause.
           ((NOT END))
                                                        ;;; Otherwise, remove any adverbs and
                                                        ;;; make secondary clauses with them.
           ((setq advrb (cadr (memass 'eq 'adverb subphrase)))
            (setq att (car advrb))
            (setq val (cdr advrb))
            (cond
              ((setq temp (assq val speed-check))
               (setq val (cdr temp))))
            (cond
              ((setq temp (assq val direction-check))
               (setq val (cdr temp))))
            (setq curclause (append curclause (list
                              (list 'is-same matt att val default-confidence))))))))
        (return T)))

(defun find-action (subphrase)
  (prog (temp mod rel)
        (setq mrel (find-after 'predicate 'active-verb subphrase))
        (cond
          ((setq temp (assq mrel action-convert))
           (cond
             ((eq (cdr temp) 'assign)
              (cond
                ((setq mod (assq 'level subphrase))
                 (setq rel (intern (format nil "~D-~D-VALUE" (cdr temp) (cdr mod)))))
                (T
                 (setq rel 'assign-value))))
             (T
              (setq rel (intern (format nil "~D-VALUE" (cdr temp))))))
           (setq mrel 'is-same)
           (setq mval rel)))))


;;; Ending uses the sentence-type of the statement to determine what type of word should be the ending.
;;; It also checks the end of the sentence for a confidence-level.  If it locates one it sets the global
 variable "mconf."
```

Appendix H

```
(defun ending (phrase)
  "Ending finds the word that should be placed in the global variable mval, and sets this
variables appropriately.  It may change the value of mconf if it locates a confidence-level
indicator at the end of a sentence."
  (prog (subphrase subsub temp unit val end)
        (setq subphrase (memass 'eq 'predicate phrase))

                                                    ;;; asking what type of ending to expec

        (setq end (car (send (eval sentence-type) :ending)))
                                                    ;;; The ending is a noun
         (cond
           ((eq (cdr end) 'noun)
            (setq mval (find-after (car end) (cdr end) subphrase))
            (SETQ PVAL MVAL)
            (setq subsub subphrase)
            (extract-adj subsub 'noun mval))
                                                    ;;; The ending is an adjective
                                                    ;;; or perhaps an adverb
           (T
            (cond
                                                    ;;; handles adjectives first
              ((setq SUBSUB (memass 'eq 'adjective subphrase))
               (setq mval (cdadr subsub))
               (SETQ PVAL MVAL)
                                                    ;;; Changes the representation of
                                                    ;;; numeric measurements
               (cond
                 ((setq temp (assq 'unit subsub))
                  (cond
                    ((setq unit (cdr (assq (cdr temp) unit-check))))
                    (T
                     (setq unit (cdr temp))))
                  (setq mval (intern (format nil "~D~D" mval unit)))
                  (SETQ PVAL MVAL)))

                                                    ;;; Sets the global attribute matt to
                                                    ;;; the class of the adjective if the
                                                    ;;; variable hasn't yet been set
               (cond
                 ((eq matt 'attribute)
                  (setq matt (caadr subsub))))
                                                    ;;; Eliminates the two conses that
                                                    ;;; refer to the adjective
               (setq subsub (delq (car subsub) subsub))
               (setq subsub (delq (car subsub) subsub))
               (cond
                 ((memass 'eq 'adjective subsub)
                  (extract-adj subsub nil mobj))))

                                                    ;;; Looks for an adverb ending
             ((setq subsub (memass 'eq 'adverb subphrase))
              (setq mval (cdadr subsub))
              (SETQ PVAL MVAL)
              (cond
                ((eq matt 'attribute)
                 (setq matt (caadr subsub))))))))

                                                    ;;; Looks for a confidence level  clause
         (cond
           ((setq val (find-after 'predicate 'clevel subphrase))
            (setq mconf val)))
         (return T)))
```

```
;;; Extract-adj finds all the qualifiers before the word contained in "before" and creates a
;;; frame using the relation "is-same," the word modified as the object, the classification
;;; of the modifier as the attribute and the modifier itself as the value.  The confidence-level
;;; is always 100.  After the frame has been created the modifier is removed from the
;;; statement to prevent it from being "found" again.

(defun extract-adj (subphrase before modified)
  (prog (adj unit-type temp att val)
        (loop while (setq adj (find-before before 'adjective subphrase)) do
          (cond
            ((eq (car adj) 'number)
             (setq unit-type (find-after 'number 'unit subphrase))
             (cond
               ((setq temp (assq unit-type unit-check))
                (setq unit-type (cdr temp))))
             (setq val (intern (format nil "~D~D" (cdr adj) unit-type)))
             (cond
               ((setq temp (find-after 'adjective 'quality subphrase))
                (setq att (cdr (assq temp quality-check)))
                (setq subphrase (delq (assq 'quality subphrase) subphrase))
                (setq subphrase (delq (assq 'adjective subphrase) subphrase)))
               ((setq temp (assq unit-type unit-convert))
                (setq att (cdr temp)))
               (T
                (setq att 'measure)))
             (setq curclause (append curclause
                                (list (list 'is-same modified att val default-confidence))))
             (setq subphrase (delq (assq 'adjective subphrase) subphrase))
             (setq subphrase (delq adj subphrase))
             (setq subphrase (delq (assq 'unit subphrase) subphrase)))
            (T
             (setq curclause (append curclause
                  (list (list 'is-same modified (car adj) (cdr adj) default-confidence))))
             (setq subphrase (delq adj subphrase))
             (setq subphrase (delq (assq 'adjective subphrase) subphrase)))))
        (return subphrase)
         ))

;;; Make-rule takes the whole of the parse and controls its conversion into Lisp code.  It goes
;;; through the parse by removing the first element from the parse list.  If this is a simple-statement
;;; then depending on its sentence-type one or more of Subject, Predicate and Ending are called
;;; to identify the various parts of the parse.  The routine starts out assuming that all of the
;;; simple-statements found are in the premise clause. When the routine encounters "Action"
;;; as the next element on the list then this default assumption is changed and the remaining
;;; statements are processed as action statements.

(defun make-rule (e-rule)
  "Make-rule controls the translation of the parse-list into lisp code. It outputs
the completed lisp rule."
  (prog (premise action substmt subnum default-relation premise? )
       (setq premise (list 'is-and))
       (setq action (list 'is-and))
       (setq subnum 1)
       (setq premise? T)
       (setq default-relation 'is-and)
       (set-values-default)
       loop1
       (setq mrel default-relation)
       (setq mconf default-confidence)
       (setq substmt (car e-rule))
       (setq e-rule (cdr e-rule))
       (cond
         (substmt
          (cond
            ((equal (car substmt) (cons 'simple-stmt subnum))
             (setq substmt (killer substmt))
             (setq sentence-type (classify substmt))
             (setq fnum 1)
             (cond
               (premise?
```

```
            (setq mrel 'is-same)
            (cond
              ((assq 'declarative substmt)
               (subject (ldiff substmt (memass 'eq 'predicate substmt)))))
            (predicate substmt)
            (ending substmt)
            (cond
              ((eq mobj 'object)
               (setq mobj matt)
               (setq matt 'attribute)))
            (setq frame1 (list mrel mobj matt mval mconf))
            (setq curclause (append (list default-relation)  (append (list frame1)
                                                                     curclause)))
            (setq premise (append premise (list curclause)))
            (setq curclause nil)

            (setq subnum (add1 subnum)))
           (T
            (cond
              ((assq 'declarative substmt)
               (subject (ldiff substmt (memass 'eq 'predicate substmt)))
               (ending substmt)
               (predicate substmt))
              ((assq 'copula-verb substmt)
               (setq mobj matt)
               (subject substmt)
               (predicate substmt))
              (T
               (subject substmt)
               (find-action substmt)))
            (cond
              ((eq mobj 'object)
               (setq mobj matt)
               (setq matt 'attribute)))
            (setq frame1 (list mrel mobj matt mval mconf))
            (setq curclause (append (list default-relation)  (append (list frame1)
                                                                     curclause)))
            (setq action (append action (list curclause)))
            (setq curclause nil)
            (setq subnum (add1 subnum)))))
       ((eq (car substmt) 'action)
        (setq premise? nil)
        (setq default-relation 'conclude)
        (set-values-default)))
      (go loop1))
     (T
      (return (list premise action))))))

;;; Set-values-default sets the defaults for some of the global variables whose values  may change frequ
ently

(defun set-values-default ()
  (prog ()
        (setq mrel nil)
        (setq mobj 'object)
        (setq matt 'attribute)
        (setq mval 'value)))


;;; Killer gets rid of all the deadwood in the parse tree.

(defun killer (word-array)
   (prog(getum)
        loop
        (cond
         ((setq getum (rassoc 'd word-array))
          (setq word-array (delq getum word-array)))
         (T
          (return word-array)))
        (go loop)))
```

Appendix H

```
;;; find-after finds the first occurance of "part" after the sentence-structure "after"

(defun find-after (after part inlist)
    (prog ()
      (setq inlist (memass 'eq after inlist))
        (setq after (cdr (assq part inlist)))
        (return after)))

;;; find-before finds the first occurance of "part" before the sentence-structure "before"

(defun find-before (before part inlist)
   (prog ()
     (return (cadr (memass 'eq part (ldiff inlist (memass 'eq before inlist)))))))

;;;  The following declaration and functions translate the lisp code generated
;;; by the translator back into English, thus completing the whole        circle.

(DEFVAR pnum 0)

(defun print-rule (rule)
  (prog ( prem conc)
        (setq pnum 0)
        (setq prem (car rule))
        (setq conc (cdr rule))
        (format T "~% IF ")
        (print-clause (cdr prem) T)
        (format T " ~% THEN ")
        (print-clause (cdar conc) nil)))


(defun print-clause (clause premise?)
  (prog (inc indent  subsub verb)
        (dolist (subphrase clause)
          (setq pnum (add1 pnum))
          (setq inc 0.1)
          (setq indent "")
          (setq subsub (cdr subphrase))
          (dolist (sentence subsub)
            (setq sentence (car subsub))
            (setq subsub (cdr subsub))
            (cond
              ((eq (first sentence) 'is-same)
               (cond
                 (premise?
                  (setq verb 'is))
                 (T
                  (COND
                    ((NEQ INC 0.1)
                     (SETQ VERB 'MATCH))
                    (T
                     (setq verb 'set))))))
              (T
               (setq verb (first sentence))))
            (cond
              (premise?
               (format T "~d ~d) The ~d of the ~d ~d" indent (plus inc pnum)
                       (third sentence) (second sentence) verb (fourth sentence)))
              (T
               (format T "~D ~d) ~D the  ~d of the ~d to  ~d" indent (plus inc pnum) verb
                       (third sentence) (second sentence)  (fourth sentence))))
            (cond
              ((neq (fifth sentence) 100)
               (format T " with confidence ~d~%    " (fifth sentence)))
              (T
               (format T " ~%    ")))
            (setq indent "   ")
            (setq inc (plus inc 0.1))))))
```

UNCLASSIFIED

Security Classification

DOCUMENT CONTROL DATA - R & D

(Security classification of title, body of abstract and indexing annotation must be entered when the overall document is classified)

1. ORIGINATING ACTIVITY DEFENCE RESEARCH ESTABLISHMENT ATLANTIC	2a. DOCUMENT SECURITY CLASSIFICATION UNCLASSIFIED 2b. GROUP

3. DOCUMENT TITLE

Circe: A Natural Language Rule Translation System for Intersensor

4. DESCRIPTIVE NOTES (Type of report and inclusive dates) TECHNICAL COMMUNICATIONS

5. AUTHOR(S) (Last name, first name, middle initial)

ELLIS, JAMES R., and DENT, C. Ann

6. DOCUMENT DATE NOVEMBER 1985	7a. TOTAL NO. OF PAGES 47	7b. NO. OF REFS 12
8a. PROJECT OR GRANT NO.	9a. ORIGINATOR'S DOCUMENT NUMBER(S) DREA TECHNICAL COMMUNICATION 85/314	
8b. CONTRACT NO.	9b. OTHER DOCUMENT NO.(S) (Any other numbers that may be assigned this document)	

10. DISTRIBUTION STATEMENT

DISTRIBUTION STATEMENT A
Approved for public release
Distribution Unlimited

11. SUPPLEMENTARY NOTES	12. SPONSORING ACTIVITY

13. ABSTRACT

Circe is a set of functions written in Lisp as an experiment in natural language processing. It was developed for knowledge acquisition by the Intersensor system, a sonar knowledge-based inference system. The main function of Circe inputs from either the keyboard or a file a rule written in English and converts it into a Lisp-like data structure. This conversion takes place in two steps: the English text is first parsed, and then the relevant information is extracted from the resulting list and assembled into rules. These rules are then written to a file where they can be used directly by the inference mechanism of Intersensor without further processing.

The paper briefly reviews prior research at DREA in this area, and contrasts the current system with its predecessors. Implementation problems are discussed, and suggestions for improvements are given.

DSIS
78-078

KEY WORDS

NATURAL LANGUAGE PROCESSING
SYNTACTIC PARSING OF NATURAL LANGUAGE
AUTOMATIC TRANSLATION OF NATURAL LANGUAGE RULES
NATURAL LANGUAGE INTERFACES FOR KNOWLEDGE ACQUISITION
OBJECT-CENTERED PROGRAMMING

INSTRUCTIONS

1. ORIGINATING ACTIVITY: Enter the name and address of the organization issuing the document.

2a. DOCUMENT SECURITY CLASSIFICATION: Enter the overall security classification of the document including special warning terms whenever applicable.

2b. GROUP: Enter security reclassification group number. The three groups are defined in Appendix 'M' of the DRB Security Regulations.

3. DOCUMENT TITLE: Enter the complete document title in all capital letters. Titles in all cases should be unclassified. If a sufficiently descriptive title cannot be selected without classification, show title classification with the usual one-capital-letter abbreviation in parentheses immediately following the title.

4. DESCRIPTIVE NOTES: Enter the category of document, e.g. technical report, technical note or technical letter. If appropriate, enter the type of document, e.g. interim, progress, summary, annual or final. Give the inclusive dates when a specific reporting period is covered.

5. AUTHOR(S): Enter the name(s) of author(s) as shown on or in the document. Enter last name, first name, middle initial. If military, show rank. The name of the principal author is an absolute minimum requirement.

6. DOCUMENT DATE: Enter the date (month, year) of Establishment approval for publication of the document.

7a. TOTAL NUMBER OF PAGES: The total page count should follow normal pagination procedures, i.e., enter the number of pages containing information.

7b. NUMBER OF REFERENCES: Enter the total number of references cited in the document.

8a. PROJECT OR GRANT NUMBER: If appropriate, enter the applicable research and development project or grant number under which the document was written.

8b. CONTRACT NUMBER: If appropriate, enter the applicable number under which the document was written.

9a. ORIGINATOR'S DOCUMENT NUMBER(S): Enter the official document number by which the document will be identified and controlled by the originating activity. This number must be unique to this document.

9b. OTHER DOCUMENT NUMBER(S): If the document has been assigned any other document numbers (either by the originator or by the sponsor), also enter this number(s).

10. DISTRIBUTION STATEMENT: Enter any limitations on further dissemination of the document, other than those imposed by security classification, using standard statements such as:

(1) "Qualified requesters may obtain copies of this document from their defence documentation center."

(2) "Announcement and dissemination of this document is not authorized without prior approval from originating activity."

11. SUPPLEMENTARY NOTES: Use for additional explanatory notes.

12. SPONSORING ACTIVITY: Enter the name of the departmental project office or laboratory sponsoring the research and development. Include address.

13. ABSTRACT: Enter an abstract giving a brief and factual summary of the document, even though it may also appear elsewhere in the body of the document itself. It is highly desirable that the abstract of classified documents be unclassified. Each paragraph of the abstract shall end with an indication of the security classification of the information in the paragraph (unless the document itself is unclassified) represented as (TS), (S), (C), (R), or (U).

The length of the abstract should be limited to 20 single-spaced standard typewritten lines; 7½ inches long.

14. KEY WORDS: Key words are technically meaningful terms or short phrases that characterize a document and could be helpful in cataloging the document. Key words should be selected so that no security classification is required. Identifiers, such as equipment model designation, trade name, military project code name, geographic location, may be used as key words but will be followed by an indication of technical content.

END

FILMED

2-86

DTIC