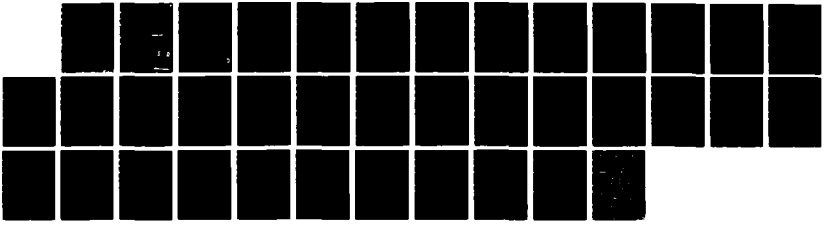
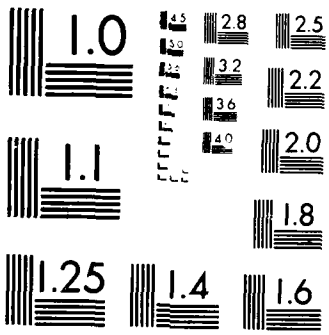
AD-A192 265 NARRATIVE DOMAIN IMPLEMENTATION(U) JAYCOR VIENNA VA 1/1
JUN 88 N00014-85-C-2044

UNCLASSIFIED F/G 5/2 NL




1.0
1.1
1.25
1.4
1.6
1.8
2.0
2.2
2.5
2.8
3.2
3.6
4.0
4.5

NARRATIVE DOMAIN IMPLEMENTATION

JAYCOR

1608 Springhill Road
Vienna, Virginia 22180-2270

4

NARRATIVE DOMAIN IMPLEMENTATION

June 1987

Report Prepared by

JAYCOR
1608 Spring Hill Road
Vienna, VA 22180-2270

Under

NRL Contract N00014-85-C-2044
Deliverable B004

For

Naval Center for Applied Research in Artificial Intelligence
Naval Research Laboratory
4555 Overlook Avenue, SW
Washington, DC 20375

TABLE OF CONTENTS

1. Introduction

The following report constitutes the deliverable for Task 4 of contract #N00014-85-C-2044. The task for this contract was to survey available knowledge representation languages and tools and assist NRL researchers in evaluating and implementing the chosen representation for the CASREP SUMMARY project. The final deliverable, as stated in the contract, requires a technical report describing the features and functionality of the developed system. This document will serve as this technical report.

In Section 2, entitled "Knowledge Representation Issues", we will describe the problem as we viewed it and some of the factors which we believed to be important in our decision on a knowledge representation tool. We also discuss our rationale for choosing the system that we did, based on the factors we believe are important. Also, in this section, we present a brief discussion on the nature of the task as we perceived it and make clear our reasoning.

The second section of this report describes the functionality of the developed system. We also describe the implementation of the knowledge representation and the behavior of the system, in general.

2. Knowledge Representation Issues

2.1. The Application

The SUMMARY expert system was developed to "summarize" (pick out particular pieces of information) Navy Casualty Report (CASREP) narrative sections. This task of summarization requires several steps which we will describe here, in a cursory manner. The first step is to obtain a normalized parse of each sentence in the message. The second step is to transform each parse into an **Information Format**[1]. These information formats are then passed to the SUMMARY expert system where rules representing general linguistic rules of summarization, domain-specific rules of classification, and rules of emphasis on particular types of information are applied to the formats. The format(s) with the highest emphasis (score) is then chosen as the "summary". In this report, ~~our~~ concern lies in the third phase of this process where summaries are chosen by the SUMMARY expert system.

2.2. Knowledge Representation Methods

There are several different methods for representing knowledge. Formal logic, procedural representations, semantic nets[2] , production systems[3] and frames are most of the prominent, broadly applicable methods of representing knowledge. In the following section, we will briefly mention the most important deficiency or advantage associated with each method. This is a general discussion of these methods and many points have been omitted in favor of including those that were relevant to our decision of choosing the best representation for the CASREP

SUMMARY project.

Purely logical representations are problematic in that systems using this type of representation are usually plagued by combinatorial explosions during the resolution process. Procedural representations are useful for some applications, but have a tendency to obscure the knowledge that they embody. Semantic nets are a useful formalism for reducing search spaces because they encode related information with direct links in the network. Associations can be made by traversing these nets, thus avoiding blind searching. This representation is particularly useful for representing ISA and SUBSET relations[4]. Production systems are useful for representing information which is naturally posed in the form of condition-action rules. This method of representation makes explicit the condition of each rule and minimizes the interaction of individual rules with one another, in theory. This method also encourages modularity through the use of independent and separate sets of rules (rule bases).

Frame-based systems have some of the attributes of procedural representations and semantic networks. A frame-based representation enables chunks of related information to be stored together in a structure called a **frame**. Frames are allowed to inherit information from higher level frames in a network. It is also possible to create super- and sub-type frames to implement relationships such as ISA and SUBSET. This method, in addition to allowing the storage of specific pieces of information in the frame slots, allows procedural information to be associated with individual frames or sets of frames. Thus, it also encompasses most of the functionality of a purely procedural representation.

2.3. The Representation Requirements of the Application

In the reimplementation of the system, a new domain model was added[5]. This domain model represented pieces of equipment in and immediatedly connected to the *starting air compressor* onboard some Navy ships. This model required a way to represent facts about each piece of equipment in an organized manner. It was also necessary to be able to compare pieces of equipment mentioned in the narrative to the model in order to determine which piece of equipment is being referenced. Another requirement of the system was the ability to infer **implicit** assertions of causality in the failure message.

The information which is used to derive a "summary" has been previously represented in the form of if-then rules in OPS-5. Although it is possible to represent this information in a number of ways, at this point in the project, we have maintained this initial representation using production rules.

The third major type of information which was to be included in the system, was the actual narrative. This information is received in the form of *information formats*. Information formats are a specialized type of knowledge representation, and therefore our task really was to implement this representation with one of the systems. See section 2.8 for a more detailed discussion of information formats as a representation.

2.4. More About Frames and Production Systems

As we mentioned before, frames are a useful method for representing related information in an organized manner. Inheritance provides efficiency in representation and implementation since common information does not have to be

respecified with every instance or subclass of the frame.

The procedures which can be attached to frames are called **daemons**. These daemons can be used to specify actions which are to be performed based on the presence, absence or even the reference to particular values contained in the frame. This facet of a frame-based system provides the problem-solving behavior. With this capability, it can be seen that any problem which can be attacked with a rule-based system can also be attacked with a frame-based system. The choice between the two (if a choice is necessary) is made based on convenience and the clarity that the particular approach will lend to the task.

If a choice between a frame-based system and a rule-based system which does not incorporate the frame methodology must be made, then the particular application must be considered to make the decision. That is, some problems lend themselves more readily to rule-based representations because the facts which must be represented are more or less atomic. Problems like this usually consist of a multitude of varied facts about a particular situation, and the rules are used to match the existence of certain facts and the perform some desired action. This action is usually one of adding a new fact or deleting an old fact. The set of facts, which are represented in the rule-systems internal representation work best if they are relatively unstructured. This is generally what this type of rule system (one without frames integrated in it) expects. When an attempt to use structured facts is made, unexpected behavior is usually exhibited in that any modification of the structured fact can re-enable rules which have previously fired, but still match another portion of the structured fact.

Frame-based systems are more oriented towards operations on particular slots. Methods are usually attached to individual slots (although it is usually possible to perform actions based on the creation or deletion of an entire frame). In this way, a daemon can be created which is designed to handle a specific condition on the particular slot (or a combination of slots and other special conditions). Because of this different way of viewing the "facts" in the system, it is easier to work with structured information without experiencing the surprising behavior of the "unstructured fact" rule-systems.

2.5. Our Choice of Representation Schemes

Frame-based systems offer features which we find useful for representing knowledge about the domain, in particular the domain model of equipment. Knowledge about how to use this domain model to resolve complex equipment references and to infer implicit causality is represented in a mixture of procedures and rules. We also found it convenient to store information-formats in frame structures so that related information could be stored together. The information which is used to classify and choose a summary is most conveniently represented as if-then rules. It was therefore required that our choice of systems allow the integrated use of frames, if-then rules, and procedures.

2.6. What Representation Schemes Were Available

The original prototype SUMMARY system made an attempt to integrate structured facts (information formats) and if-then rules using the OPS-5 rule system and built-in fact representation scheme. However, the integration of pro-

cedural information, not in the form of if-then rules, was not straightforward. FRL (Frame Representation Language)[6] was available, but it did not incorporate a production rule system. The only other choice available to us, at the time, was a rule system called YAPS (Yet Another Production System)[7] which ran in the University of Maryland Franz Lisp Environment[8]. As part of the UoM environment, flavors (a frame-like, object oriented package) had been included.

2.7. The Choice of YAPS

With these three alternatives, we made the decision to implement the new version of the system using the UoM YAPS and flavors systems. Together, they offered access to a rule system and a frame-based system all at once, although the two are not truly integrated. This solution gave us the capability to develop a system which is able to make use of both methodologies. YAPS gave us the capability to manipulate the rule system from higher level lisp routines, integrate arbitrary procedures in the rules if necessary, and access (although, not gracefully) flavor objects. This capability allowed us to interface the domain model, which is implemented in flavors, to the YAPS rule system.

2.8. The Nature of the Task w.r.t. Information Formats

From time to time, there seems to have been some confusion over what is a knowledge representation and what is an implementation of a knowledge representation. This discussion pertains in particular to the information format - frame relationship. Information formats **are** a knowledge representation. They are a **special-purpose** knowledge representation of information which was conveyed in

a narrative, in our case. Information formats were initially used for information retrieval purposes in the Linguistic String Project at NYU[1].

Information formats have a tree-like structure. Subnodes of the tree represent semantic categories which have some relevant meaning assigned to them. At the leaves of the tree, are the actual words contained in the original text. There exists a inherent method for interpreting this knowledge representation. That is, the knowledge contained in these structures is defined by the procedures which generated the structure.

Since we view information formats as being a knowledge representation, we did not interpret this task as one of developing another knowledge representation, but one of implementing this representation. We chose a frame based representation, not because information formats **are** frames, but because frames are convenient. In fact, information formats do not make use of anything frame-like other than that one can think of semantic categories as slots in the frame. Frames were a convenient way of keeping the related information together. The implementation of the domain model using frames is a much clearer example of how frame representations are used. This model made use of inheritance and procedural daemons in addition to the slot-grouping aspect of frames.

3. Features and Functionality of the System

3.1. What Was Actually Implemented

In the version of the system which is to serve as the deliverable for this task, no new additions to the knowledge bases or information formats were included. The domain model, developed by Ken Wauchope, was designed using the methodology mentioned in the previous sections, but it is not included in this report and is not fully compatible with this particular version of the system. Some slight changes to the way the rule system is run were made when the domain model was added. In particular, that version of the system uses the goal directed strategy for applying rules, where the version described in this report, uses forward chaining and separate rule bases. The reason that we do not describe the goal directed version as part of this report, is that it was our purpose to closely duplicate the behavior of the OPS-5 version of the system.

In the following sections, we will describe the flavor implementation of information formats, the YAPS implementation of the rules from the previous OPS-5 version, and a few changes that were made in that transition. The code for this system resides on the NCARAI's vax 11/780 in the directory /usr/aic/daley/natlang/krep/Oldsumsys/Plainyaps. It is run in the Maryland Lisp Environment with the YAPS system loaded in. A listing of te code will be provided as an appendix to this report.

3.2. The Implementation of Information Formats

A set of flavor definitions resides in the file *defflavors.l*. The following is a

list of the flavor objects used to implement information formats: HIERARCHY-INFO, ENTRY, FORMAT, CONN. We include a brief description of each flavor, any associated procedures, and a statement of what the function of this flavor is.

HIERARCHY-INFO is a flavor which is defined solely to be mixed in with the other flavors FORMAT, CONN and ENTRY. It has one slot called "parent". When this flavor is mixed in with another flavor, the effect will be to add that slot. This additional slot enables us to retrieve facts based on elements in the slots of a child structure and follow its parent pointer up one level and access the actual parent structure, if necessary. When a flavor object is completed, a method associated with that flavor is executed and it causes the parent object to record itself in all of its children's **parent** slot.

ENTRY corresponds to table entries in a format table. It actually houses the text and can house individual modifiers such as negations. The following is a list of slots and their purpose. It should be noted that these slots constitute only a small number of the possible number of slots, but the addition of these other slots will be performed as work under another contract.

-**head** contains the head word of the phrase that this particular entry represents
-**text** contains a list of the entire phrase or entry
-**headcat** is used by the rules to classify the item in the head slot
-**neg** contains any negation words modifying the head element
-**code** contains the code number of pieces of equipment
-**modal** contains any modality modifiers to the head slot
-**det** contains any determiners modifying the head element

The ENTRY flavor is used to fill slots of the CONN and FORMAT flavors where actual blocks of text are contained. Associated with the ENTRY flavor is the procedure called **change-score**. This method allows a rule to tell an ENTRY flavor

that it is incrementing the ENTRY's parent flavor by some amount. This method operates differently from the other change-score methods, in that it does not change the score of the ENTRY (which doesn't have a score slot anyway), it just passes the score up to its parent flavor via the **parent** slot.

FORMAT is the flavor which corresponds to a single line in the format table. It has slots for every possible semantic category that is defined in the particular sublanguage. The following is a list of categories in this application: **signal, repair, process, part, org, func, assist, invest, msg, procure, property, stask** and **status.** In addition to these slots, a **score** slot is used by the rule system and a **parent** slot is included in order to facilitate upward traversal of nested structures in the case of a connective. The slots named **sentenceno, formatno** and **seqnum** are used to number assertions sequentially. These slots are not used by this version of the system, but were included to facilitate future expansion. The FORMAT structure is the representation of simple assertions.

There is a group of procedures called **methods** associated with the FORMAT flavor. One group of methods with names constructed by appending **set-** to the slot name, are run any time the slot is accessed. These methods are responsible for performing type checking. Another method called **set-childrens-parent** is used to install a pointer in every slot with an ENTRY filling it, to the parent structure. **make-text-field** is a method which collects all the text fields in a FORMAT, concatenates them together and then prints the text. This is used to print the FORMAT which is found to have the highest score at the end of processing. The last method associated with FORMAT is the **change-score** method

which is triggered any time the **score** slot is referenced. It checks to see that all arguments are numbers, adds the number and then sets the **score** slot to the sum.

Any entry in the CONN column of a format table will generate a CONN structure. A **CONN** flavor connects two other flavors which may in turn be CONN flavors. These two arguments are actually implemented as slots named **arg1** and **arg2** containing pointers to the embedded flavors. CONN has a slot called **op** which is filled by an ENTRY flavor. This slot contains the ENTRY flavor which contains the actual connecting phrase or word. CONN also has a **score** and **parent** slot. The score slot is used by the scoring rules as it is in FORMAT. The **parent** slot is used to point to any higher level structure to which this structure is an argument (embedded CONN's). CONN has only one slot with a type-checking method associated with it (the **op** slot). The **change-score** method and the **set-childrens-parent** method work exactly like FORMAT's corresponding methods do.

3.3. Differenced From the OPS-5 Version

The methods called **switcheroo** and **propagate-score** associated with CONN perform functions which were previously performed by rules in the OPS-5 version of the system. **switcheroo** is called by a rule in the rule base to "normalize" the order of causal assertions (ex: A due-to B ==> B cause A). This function just switches the value of the **arg1** slot with the value of the **arg2** slot. **propagate-score** is not called from the rule system. It is attached to the score slot of the CONN flavor as a method. Whenever an attempt to change a score is made, this method is triggered. Propagate-score causes the higher level CONN

which has just been scored to add the amount of that score to its children and keep passing the score down until only FORMATs are left to receive the score. After passing down the score, the CONN zeros its own score. This was previously done by the **Bequeath** rule in the OPS-5 version. We found this to be a more appropriate function for a method rather than a rule.

3.4. The Behavior of the Rule System

The rules for the SUMMARY system reside in a file named *yapsrules.l*. Although all of the rules are contained in one file, when the system is actually running, there are four distinct rule bases. This is achieved via the *(usc-yaps-db DBNAME)* statements distributed throughout the rule code. Whenever the yaps interpreter encounters one of these statements, all rules that are added afterward are associated with the rule base name by the argument DBNAME. The name of the rule bases used in this system are **CATEG, INFER, SCORE** and **TALLY**.

In order to execute the rule base, it is necessary to convert a set listified table entries into flavor objects. This can be accomplished by a program called **make-facts** which is contained in the file named *makefacts.l*. This program takes a single list of listified FORMATS and/or CONNs which represent a message, converts them into flavors and constructs a fact for each FORMAT, CONN, and ENTRY flavor created. See the documentation in the file for further description of the necessary formats. This program was developed by Ken Wauchope.

Once the flavor objects and corresponding facts have been created, it is necessary to send a list containing the facts to each of the previously mentioned rule bases, in order. Order is important because each successive rule base uses the

results from the previous one. *runsum.l* is a lisp file which contains all the commands necessary to run the SUMMARY system. After loading this file into the YAPS environment, the user will be prompted for the name of a file containing the listified format table. After entering the name, all necessary files will be loaded and the rule system will be executed. At the end of the run, a rule will cause the winning format(s) to be printed out.

3.5. Rule System Mode

The conflict resolution strategy used in this rule system is the age first strategy. This strategy causes bindings with younger facts to be considered before other competing bindings. If ties occur in bindings with the same aged facts, then the specificity or length of the tied facts is used to break the tie, where the longest facts in the bindings receive preference. If ties still remain, then the conflict is resolved randomly. This rule system operates in a purely forward-chaining mode. For every cycle of the rule base, the left hand sides of all the rules in the current rule base are checked for matches to the current set of facts. If a fact or set of facts can be used to match a left hand side of a rule, then a binding is formed and the right hand side of the rule is executed. See the YAPS user manual for further information on conflict resolution.

References

1. Sager, Naomi, *Natural Language Information Processing*, Addison-Wesley Publishing Co., Reading, Mass., 1981.
2. Quillian, R., "Semantic Memory," *Semantic Information Processing*, MIT Press, Cambridge, Mass., 1968. M. Minsky (Ed.)
3. Newell, A. and Simon, H. A., *Human Problem Solving*, Printice-Hall, Englewood Cliffs, NJ, 1972.
4. Barr, Avron and Feigenbaum, Edward A., *The Handbook of Artificial Intelligence*, Vol. I, Heuristech Press, Stanford, California, 1981.
5. Wauchope, Kenneth, DiBenigno, Kathryn, and Marsh, Elaine, *Use of a Domain Model in Understanding Equipment Failure Messages*, submitted for publications, December, 1986.
6. Roberts, R. Bruce and Goldstein, Ira P., *The FRL Manual*, MIT Artificial Intelligence Laboratory, September 1977.
7. Allen, Elizabeth M., *YAPS: Yet Another Production System*, University of Maryland, Department of Computer Science, TR-1146, December 1983.
8. Allen, Elizabeth M., Trigg, Randall H., and Wood, Richard J., *The Maryland Artificial Intelligence Group Franz Lisp Environment*, University of Maryland, Department of Computer Science, TR-1226, December 1984.

APPENDICES

APPENDIX A

FILE: defflavors.1

```
; ----------------------------DEFFLAVORS.L-------------------------------

; Define the object and mixin flavors for the YAPS facts in the CASREP
; summary system.

; ------------------------------------------------------------------------
(setq *flavor-recombine-immediate-flavor-only* 't)

; ------------------------------------------------------------------------
; To be mixed in to all object flavors, to serve as a back pointer to the
; flavor object which is the parent of the current one (and of which the
; current is a slot value):

(defflavor HIERARCHY-INFO
       (parent)         nil :mix-in)
; ------------------------------------------------------------------------
(defflavor ENTRY
       (head
        text
        headcat
        neg
        code
        modal
        det)      nil   :included-flavors HIERARCHY-INFO)

; ------------------------------------------------------------------------
; making sure that item to add to score is numeric then add and set the slot
; to the new value.
(defmethod (ENTRY change-score) (value)
       (cond ((fixp value)
              (>> (<< self 'parent) 'score (+ value (<< (<< self 'parent)
                                                        'score))))
             (t (patom "ERROR: argument not a fixnum ")(terpri))))

; ------------------------------------------------------------------------
(defflavor FORMAT
       (sentenceno formatno seqnum assert-type signal repair
        process part org func assist invest msg
        procure property stask status (score 0) text)
       nil     :included-flavors HIERARCHY-INFO)

; method to collect all text fields and make a master text field in the
; FORMAT flavor.

(defmethod (FORMAT make-text-field) nil
       (prog (text-list slot-list)
         (setq slot-list '(signal repair process part org func assist invest
                                  msg procure property stask status))
         loop
               (cond (slot-list
                      (cond ((<< self (car slot-list))
                       (setq text-list
                             (append1 text-list
                                 (<< (<< self (car slot-list)) 'text)))))
                       (setq slot-list (cdr slot-list))
                       (go loop)))
```

```
                (patom "TEXT for ")(print self)(terpri)
                (print text-list)(terpri)
                (>> self 'text text-list)))



; method to establish backpointers from children to parents:

(defmethod (FORMAT set-childrens-parent) ()
  (cond ((<< self 'signal)
         (>> (<< self 'signal) 'parent self)))
  (cond ((<< self 'process)
         (>> (<< self 'process) 'parent self)))
  (cond ((<< self 'func)
         (>> (<< self 'func) 'parent self)))
  (cond ((<< self 'part)
         (>> (<< self 'part) 'parent self)))
  (cond ((<< self 'property)
         (>> (<< self 'property) 'parent self)))
  (cond ((<< self 'repair)
         (>> (<< self 'repair) 'parent self)))
  (cond ((<< self 'assist)
         (>> (<< self 'assist) 'parent self)))
  (cond ((<< self 'invest)
         (>> (<< self 'invest) 'parent self)))
  (cond ((<< self 'procure)
         (>> (<< self 'procure) 'parent self)))
  (cond ((<< self 'org)
         (>> (<< self 'org) 'parent self)))
  (cond ((<< self 'msg)
         (>> (<< self 'msg) 'parent self)))
  (cond ((<< self 'status)
         (>> (<< self 'status) 'parent self)))
  (cond ((<< self 'stask)
         (>> (<< self 'stask) 'parent self))))

; we are defining our own instance-variable-setting methods so as to
; perform strong type checking:

; strong type checking
; This is a macro which is used in defining the code for methods attached to
; value setting conditions on the slots of the FORMAT entries.  It will
; insure that all FORMAT slots are filled in with ENTRY type formats.

(defmacro check-flavor-type (slot-name slot-type value)
 (let ((type (instancep ,value)))
        (cond ((eq type ,slot-type)
                 (>> self ,slot-name ,value))
              (t
                (patom "ERROR: **")(print ,value)
                (patom "** not a flavor instance of ")
                (patom ,slot-type)(terpri)
              ))))

(defmethod (FORMAT set-signal) (value-to-set)
```

```
        (check-flavor-type 'signal 'ENTRY value-to-set))
(defmethod (FORMAT set-process) (value-to-set)
        (check-flavor-type 'process 'ENTRY value-to-set))
(defmethod (FORMAT set-func) (value-to-set)
        (check-flavor-type 'func 'ENTRY value-to-set))
(defmethod (FORMAT set-part) (value-to-set)
        (check-flavor-type 'part 'ENTRY value-to-set))
(defmethod (FORMAT set-property) (value-to-set)
        (check-flavor-type 'property 'ENTRY value-to-set))
(defmethod (FORMAT set-repair) (value-to-set)
        (check-flavor-type 'repair 'ENTRY value-to-set))
(defmethod (FORMAT set-assist) (value-to-set)
        (check-flavor-type 'assist 'ENTRY value-to-set))
(defmethod (FORMAT set-invest) (value-to-set)
        (check-flavor-type 'invest 'ENTRY value-to-set))
(defmethod (FORMAT set-procure) (value-to-set)
        (check-flavor-type 'procure 'ENTRY value-to-set))
(defmethod (FORMAT set-org) (value-to-set)
        (check-flavor-type 'org 'ENTRY value-to-set))
(defmethod (FORMAT set-msg) (value-to-set)
        (check-flavor-type 'msg 'ENTRY value-to-set))
(defmethod (FORMAT set-stask) (value-to-set)
        (check-flavor-type 'stask 'ENTRY value-to-set))


; ------------------------------------------------------------------------
; making sure that item to add to score is numeric then add and set the slot
; to the new value.
(defmethod (FORMAT change-score) (value)
        (cond ((fixp value)
                (>> self 'score (+ value (<< self 'score))))
              (t (patom "ERROR: argument not a fixnum ")(terpri))))


; ------------------------------------------------------------------------
(defflavor CONN (op sentenceno seqnum arg1 arg2 (score 0)) nil
                :included-flavors HIERARCHY-INFO)

(defmethod (CONN set-op) (value-to-set)
        (check-flavor-type 'op 'ENTRY value-to-set))

; making sure that item to add to score is numeric then add and set the slot
; to the new value.
(defmethod (CONN change-score) (value)
        (cond ((fixp value)
                (>> self 'score (+ value (<< self 'score))))
              (t (patom "ERROR: argument not a fixnum ")(terpri))))


(defmethod (CONN propagate-score) ()
; method that does what the OPS5 score propagation rule does
        (cond ((and
                (<< self 'arg1)
                (<< self 'arg2)
                (neq (<< self 'score) 0))
                        (<- (<< self 'arg1) 'change-score (<< self 'score))
```

```
                        (<- (<< self 'arg2) 'change-score (<< self 'score))
                        (>> self 'score 0))))

(defmethod (CONN set-childrens-parent) ()
        (cond ((<< self 'op)
                (>> (<< self 'op) 'parent self)))
        (cond ((<< self 'arg1)
                (>> (<< self 'arg1) 'parent self)))
        (cond ((<< self 'arg2)
                (>> (<< self 'arg2) 'parent self))))

; this method switches the arg1 and arg2 of a CONN (as in rule DUE_TO):

(defmethod (CONN switcheroo) ()
        (let (( x (<< self 'arg1)))
                (>> self 'arg1 (<< self 'arg2))
                (>> self 'arg2 x)))
```

APPENDIX B

FILE: yapsrules.1

```
(setq CATEG (make-instance 'yaps-database))
(setq INFER (make-instance 'yaps-database))
(setq SCORE (make-instance 'yaps-database))
(setq TALLY (make-instance 'yaps-database))

; ------------------------CATEGORIZATION RULES-----------------------------------
(use-yaps-db CATEG)

(p catgz-bad
     (status -status_instance)
test (null (<< -status_instance 'headcat))
     (memq (<< -status_instance 'head)
        '(abort aborted bad bent break broken burn chip clog clogged
          corrode corroded crack damage drop erosion erratic erratically
          error fail failure fault faulty improper improperly inadequate
          inop inoperative lack leak lose loss lost loud low malfunction
          material poor scour sheared slippage slow split spot unable
          unusable vibration warp warped wear wiped worn wrong))
-->  (>> -status_instance 'headcat 'bad))

(p catgz-cause
     (op -op_instance)
test (null (<< -op_instance 'headcat))
     (memq (<< -op_instance 'head)
        '(affect cause make produce render result))
-->  (>> -op_instance 'headcat 'cause))

(p catgz-code
     (part -part_instance)
test (null (<< -part_instance 'headcat))
     (memq (<< -part_instance 'code) '(kwr-37 ky-8))
-->  (>> -part_instance 'headcat 'assembly))

(p catgz-component
     (part -part_instance)
test (null (<< -part_instance 'headcat))
     (memq (<< -part_instance 'head)
        '(amplifier antenna apc assembly circuit driver exciter pa ppc rf
          valve regulating_valve))
-->  (>> -part_instance 'headcat 'assembly))

(p catgz-impair
     (op -op_instance)
test (null (<< -op_instance 'headcat))
     (memq (<< -op_instance 'head)
        '(impair inhibit prevent stop))
-->  (>> -op_instance 'headcat 'impair))

(p catgz-problem
     (status -status_instance)
test (null (<< -status_instance 'headcat))
     (memq (<< -status_instance 'head)
        '(damage difficulty defect failure fault malfunction problem))
-->  (>> -status_instance 'headcat 'problem))

(p catgz-show
```

```
     (op -op_instance)
test (null (<< -op_instance 'headcat))
     (memq (<< -op_instance 'head)
        '(show indicate reveal))
-->  (>> -op_instance 'headcat 'show))

; ---------------------- INFERENCING RULES -----------------------------
(use-yaps-db INFER)

(p zero-to-bad
     (status -status_instance)
test (memq (<< -status_instance 'quant) '(zero 0))
     (neq (<< -status_instance 'headcat) 'bad)
-->  (>> -status_instance 'quant nil)
     (>> -status_instance 'head 'bad)
     (>> -status_instance 'headcat 'bad)
     (>> -status_instance 'text 'bad)
     (refresh 1))

(p quant-to-bad
     (status -status_instance)
test (eq (<< -status_instance 'quant-mod) 'in_excess_of)
-->  (>> -status_instance 'quant-mod nil)
     (>> -status_instance 'head 'bad)
     (>> -status_instance 'headcat 'bad)
     (>> -status_instance 'text 'bad)
     (refresh 1))

(p neg-to-bad-status
     (status -status_instance)
test (not (null (<< -status_instance 'neg)))
     (neq (<< -status_instance 'headcat) 'bad)
-->  (>> -status_instance 'headcat 'bad)
     (refresh 1))

; CAUSE-BAD
; if there is a CONN whose OP has headcat IMPAIR,
; and all fields of ARG2 have null STATUS,
; then give one of the fields a STATUS of BAD.
; Example:  "x stopped new SAC" is not a cause-bad because NEW is
; a status.  This does not make sense.  Perhaps it should check to
; see that ARG2 does not have a status HEADCAT = BAD -- i.e. no bad
; statuses, not statuses in general.  E.g. "x stopped SAC failure"
; is a cause-good.
; The call to new-entry in the ops5 code is to generate the new fact
; that there is now a new ENTRY floating around with status bad, so
; that it is available to fire other rules within this phase (database).
; We have to do the same.

(p cause-bad
     (op -op_instance)
     (conn -conn_instance)
     (format -format_instance)
     (  (status -status_instance) with
        (eq (<< (<< -status_instance 'parent) 'parent)
            -format_instance))
```

```
test (eq (<< -op_instance 'parent) -conn_instance)
     (eq (<< -op_instance 'headcat) 'impair)
     (eq -format_instance (<< -conn_instance 'arg2))
     (or (<< -format_instance 'property)
         (<< -format_instance 'signal)
         (<< -format_instance 'part)
         (<< -format_instance 'func))
-->  (setq status_instance (make-instance 'CANT-STATUS 'head 'bad 'text 'bad
                                'headcat 'bad 'parent (or
                                (<< -format_instance 'property)
                                (<< -format_instance 'signal)
                                (<< -format_instance 'part)
                                (<< -format_instance 'func))))
     (>> (or (<< -format_instance 'property)
             (<< -format_instance 'signal)
             (<< -format_instance 'part)
             (<< -format_instance 'func)) 'status status_instance)
     (fact status ^status_instance)
     (>> -op_instance 'headcat 'cause)
     (refresh 1)
     (setq fact-list (cons (list 'status status_instance) fact-list)))

; SMOKE-FIRE
; If there is a CONN whose OP.HEADCAT = CAUSE
; and whose ARG1 has no field having a STATUS
; and whose ARG2 has a field with a STATUS.HEADCAT = BAD
; then give ARG1 a STATUS (somewhere) of BAD

(p smoke-fire
     (conn -conn_instance)
     (op  -op_instance)
     (format -arg1)
     (format -arg2)
     (status -status_instance)
     (~ (status -any_status) with
        (eq (<< (<< -any_status 'parent) 'parent) -arg1))
test (eq -op_instance (<< -conn_instance 'op))
     (eq -arg1 (<< -conn_instance 'arg1))
     (eq -arg2 (<< -conn_instance 'arg2))
     (eq (<< (<< -status_instance 'parent) 'parent) -arg2)
     (eq (<< -status_instance 'headcat) 'bad)
     (eq (<< -op_instance 'headcat) 'cause)
     (or (<< -arg1 'property)
         (<< -arg1 'signal)
         (<< -arg1 'part)
         (<< -arg1 'func))
-->  (setq new_status (make-instance 'CANT-STATUS 'head 'bad 'text 'bad
                        'headcat 'bad
                        'parent (or (<< -arg1 'property)(<< -arg1 'signal)
                                    (<< -arg1 'part)(<< -arg1 'func))))
     (>> (or (<< -arg1 'property)(<< -arg1 'signal)
             (<< -arg1 'part)(<< -arg1 'func)) 'status new_status)
     (fact status ^new_status)
     (setq fact-list (cons (list 'status new_status) fact-list)))

(p modal-move
```

```
     (conn -conn_instance)
test (memq (<< (<< -conn_instance 'op) 'head) '(suspect appear believe))
     (eq (instancep (<< -conn_instance 'arg2)) 'CONN)
-->  (>> (<< (<< -conn_instance 'arg2) 'op)
         'modal
         (<< (<< -conn_instance 'op) 'head)))

(p due-to
     (op -op_instance)
test (memq (<< -op_instance 'head) '(due_to result))
-->  (>> -op_instance 'headcat 'cause)
     (<- (<< -op_instance 'parent) 'switcheroo))

; -------------------------SCORING RULE DATABASE-------------------------------
(use-yaps-db SCORE)

(p universal
     (part -part_instance)
test (memq (<< -part_instance 'det) '(all each every))
-->  (<- -part_instance 'change-score -1))

(p assembly
     (part -part_instance)
test (eq (<< -part_instance 'headcat) 'assembly)
-->  (<- -part_instance 'change-score +2))

(p other-part
     (part -part_instance)
test (neq (<< -part_instance 'headcat) 'assembly)
     (neq (<< -part_instance 'head) 'oil)
-->  (<- -part_instance 'change-score +1))

(p any-signal
     (signal -signal_instance)
-->  (<- -signal_instance 'change-score +1))

(p bad-status
     (status -status_instance)
test (eq (<< -status_instance 'headcat) 'bad)
     (null (<< -status_instance 'neg))
-->  (<- -status_instance 'change-score +10))

(p neg-func
     (func -func_instance)
test (not (null (<< -func_instance 'neg)))
-->  (<- -func_instance 'change-score +8))

(p neg-process
     (process -process_instance)
test (not (null (<< -process_instance 'neg)))
-->  (<- -process_instance 'change-score +8))

; PROBLEM-IS
; if there is a status hanging around with headcat = problem
; and if there is a conn with op head word of 'be where the
; conn is the parent of the format containing the status = problem
```

```
; add one to the arg2 score of the conn.

(p problem-is
     (op -op_instance)
     (status -status_instance)
     (conn -conn_instance)
test (eq (<< -op_instance 'head) 'be)
     (eq (<< -status_instance 'headcat) 'problem)
     (eq (<< (<< -status_instance 'parent) 'parent)
         (<< -conn_instance 'arg1))
     (eq -conn_instance (<< -op_instance 'parent))
-->  (<- (<< -conn_instance 'arg2) 'change-score +1))

(p find
     (op -op_instance)
test (memq (<< -op_instance 'head) '(find determine discover))
-->  (<- (<< (<< -op_instance 'parent) 'arg2) 'change-score +1))

(p invest-conn
     (conn -conn_instance)
test (eq (<< (<< -conn_instance 'op) 'headcat) 'show)
     (not (null (<< (<< -conn_instance 'arg1) 'invest)))
-->  (<- (<< -conn_instance 'arg2) 'change-score +1))

(p bequeath
      (conn -conn_instance)
test  (neq (<< -conn_instance 'score) 0)
-->   (<- -conn_instance 'propagate-score)
      (refresh 1))

(p modal-obj
     (op -op_instance)
test (memq (<< -op_instance 'head) '(suspect appear believe))
-->  (<- (<< (<< -op_instance 'parent) 'arg2) 'change-score +1))

(p cause
     (op -op_instance)
test (eq (<< -op_instance 'headcat) 'cause)
     (null (<< -op_instance 'modal))
-->  (<- (<< (<< -op_instance 'parent) 'arg1) 'change-score +1))

(p cause-may
     (op -op_instance)
test (eq (<< -op_instance 'headcat) 'cause)
     (not (null (<< -op_instance 'modal)))
-->  (<- (<< (<< -op_instance 'parent) 'arg1) 'change-score +1))

; -------------------------FINAL TALLY PHASE----------------------------

(use-yaps-db TALLY)

(p add-score-facts
     (format -format_instance)
-->  (fact score ^(<< -format_instance 'score) ^-format_instance))

(p delete-smaller
```

```
     (score -value1 -format_instance1)
     (score -value2 -format_instance2)
test (< -value1 -value2)
-->  (remove 1))
```

APPENDIX C

FILE: makefacts.1

```
(eval-when (eval compile) (load 'defflavors))
(declare (special
                **ALLOWABLE-SLOTS**))
(setq **ALLOWABLE-SLOTS** `(format arg1 arg2 part status func stask
                            signal repair process org assist invest
                            msg procure property conn op))

; ------------------------------MAKEFACTS.L--------------------------------

; This is some code to generate flavor objects from listified information
; formats, and to generate a YAPS fact for certain flavor objects and their
; instance variables.

; Current object flavors are FORMAT, CONN and ENTRY.  Current facts wanted
; are formats, conns, ops, and all of FORMAT's instance variables (status,
; part, func, etc.) -- we don't need facts for arg1, arg2, or any of
; ENTRY's instance variables (head, text, det, etc.)  Information format
; components are assumed to be either of the form

; (format (<slotname1> (entry (head x)(text y)...))
;         (<slotname2> (entry ...))
;         ...)

;  or

; (conn (op (entry (head x)(text y)...))
;       (arg1 (format ...))
;       (arg2 (format ...))

; YAPS facts will be of the form (<name> <flavor-object>), such as
; (format <f1>), (status <f2>).  A list of these (the global object
; fact-list) is assembled for later loading into a YAPS database.

; If the input to be processed is a single information format line,
; run do-format-or-conn on it.  If it is a single report (list of
; format lines), run make-facts on it.  Be sure you've first initialized
; the variable fact-list -- to nil, or some initial set of facts such
; as goals, control strategy facts, etc.
; ---------------------------------------------------------------------------

(defun make-facts (report)
 (setq fact-list nil sentenceno 0 formatno 0 seqnum 0)
 (for sentence in report do
    (setq sentenceno (1+ sentenceno))
    (do-format-or-conn sentence)))


; Each report consists of a list of one or more formats and/or conns.

; ---------------------------------------------------------------------------

(defun do-format-or-conn (arg)
(prog (fact-name flavor fact)
(setq seqnum (add1 seqnum))
(setq fact-name (car arg) flavor (uppercase fact-name))
(setq fact (eval `(make-instance ',flavor ,@(do-slots (cdr arg)))))
```

```
(<- fact 'set-childrens-parent)
(>> fact 'sentenceno sentenceno)
(>> fact 'seqnum seqnum)
(cond
  ((eq fact-name 'format)
     (<- fact 'make-text-field)
     (>> fact 'formatno (setq formatno (1+ formatno)))))
(setq fact-list (cons `(,fact-name ,fact) fact-list))
(return fact)))

; Example:
; (format (part (entry (head SAC)(text SAC)(det the)))
;         (status (entry (head broken)(text (really broken)))))

; is fed into do-format-or-conn.  Type is "format", so the fact generated
; is (make-instance 'FORMAT 'part <obj1> 'status <obj2>).  (The subsequent
; arguments to the make-instance are generated and returned by do-slots,
; and spliced in.)  The fact generated by the make-instance, in this case
; '(format <fact>), is then added to the front of fact-list.  Since in the
; present system formats or conns can have other facts as children, we send
; the child backpointer insertion message to the fact.  Finally we return
; the fact itself, since at other than the top level the format/conn fact
; is needed to plug into the ARG1 or ARG2 of some other CONN.
```

```
; ------------------------------------------------------------------------

(defun do-slots (slots)
(prog (slot fact slot-name)
(cond
  ((null slots)(return nil))
  (t (setq slot (car slots) slot-name (car slot))       ; do next slot in slots
;(patom "SLOT NAME ")(print slot-name)(terpri)
     (cond
;(patom "CADR SLOT IS ")(print (cadr slot))(terpri)
       ((memq slot-name '(arg1 arg2))
          (setq fact (do-format-or-conn (cadr slot))))
       ((memq slot-name **ALLOWABLE-SLOTS**)
          (setq fact (do-entry (cadr slot)))
          (setq fact-list (cons `(,slot-name ,fact) fact-list)))
       (t (patom "ERROR: ")(print slot-name)
          (patom " not an allowable slot name")(terpri)))
     (return (cons (list 'quote slot-name)
                   (cons fact (do-slots (cdr slots))))))))))     ; do rest


; Do-slots handles the list of slots of a format or conn.  A slot can be
; an ARG1 or ARG2 (in which case it contains a single subslot of flavor
; FORMAT or CONN, and is handled by do-format-or-conn), or it is of flavor
; ENTRY (op, status, part, etc.) in which case the subslots are not flavor
; objects but just quoted literals, and are handled by do-entry.

; Example:
; ( (status (entry (head broken)(text broken)))
;   (part   (entry (head alarm)(text (LO alarm)))) )

; The first slot is stripped off and processed first.  Name = "status",
; and do-entry of the cadr generates an ENTRY object with the slots "head"
; and "text" filled in.  We add the fact '(status <fact>) to the fact-list,
; and after recursively processing the remaining slot (part...), return
; ('status <fact1> 'part <fact2>) as the return value of the function.
; This return value is then spliced in as the argument list of a make-
; instance).

; If the slot to be processed is an ARG1 or ARG2, we want to add its single
; child (format or conn) as a fact so we'll let do-format-or-conn do that
; for us.  At present we aren't interested in making the ARG a fact itself.

; Ultimately a smarter version of this code would be entirely data-driven,
; and so we wouldn't need separate do-format-or-conn and do-entry modules
; but could have one generic module to do both.  It would have to bottom out
; at ENTRY, however, since we don't want entry slots to be flavor objects/
; facts but just literal values.  Also, the data-driven routine would want
; to generate facts of the form (arg1 <format/conn>) and (arg2 <format/conn>)
; which our present system doesn't need -- but might in the future.

; ------------------------------------------------------------------------

(defun do-entry (slot)
(prog (flavor slots)
```

```
(setq flavor (uppercase (car slot)) slots (cdr slot))
;(patom "FLAVOR is ")(print flavor)(terpri)
;(patom "SLOTS ")(print slots)(terpri)
(return (eval `(make-instance ',flavor
              ,@(mapcar '(lambda (elem) (list 'quote elem))
                        (apply 'append slots)))))))

; Do-entry of slots ((head valve)(text (LO pressure valve))) for example
; does a (make-instance 'ENTRY 'head 'valve 'text '(LO pressure valve)) --
; it just inserts quotes in front of each entry in the slot list.

; ---------------------------------------------------------------------------
; Uppercase returns an atom with the same name as the argument but all caps.

(defun uppercase (atom)
(implode
   (mapcar 'ascii
           (mapcar '(lambda (char)(- char 32))
                   (substringn atom 1)))))
```

END

DATE

FILMED

6-88

DTIC