MASSACHUSETTS INSTITUTE OF TECHNOLOGY
ARTIFICIAL INTELLIGENCE LABORATORY

Working Paper 151 August 1977

AMORD

A DEDUCTIVE PROCEDURE SYSTEM

by

Johan de Kleer, Jon Doyle*,
Charles Rich, Guy L. Steele Jr.**, and Gerald Jay Sussman

Abstract:

We have implemented an interpreter for a rule-based system, AMORD, based on a non-chronological control structure and a system of automatically maintained data-dependencies. The purpose of this paper is tutorial. We wish to illustrate:
{1} The discipline of explicit control and dependencies,
{2} How to use AMORD, and
{3} One way to implement the mechanisms provided by AMORD.

This paper is organized into sections. The first section is a short "reference manual" describing the major features of AMORD. Next, we present some examples which illustrate the style of expression encouraged by AMORD. This style makes control information explicit in a rule-manipulable form, and depends on an understanding of the use of non-chronological justifications for program beliefs as a means for determining the current set of beliefs. The third section is a brief description of the Truth Maintenance System employed by AMORD for maintaining these justifications and program beliefs. The fourth section presents a completely annotated interpreter for AMORD, written in SCHEME.

* Fannie and John Hertz Foundation Fellow
** NSF Fellow

This research was conducted at the Artificial Intelligence Laboratory of the Massachusetts Institute of Technology. Support for the Laboratory's artificial intelligence research is provided in part by the Advanced Research Projects Agency of the Department of Defense under Office of Naval Research contract number N00014-75-C-0643.

Acknowledgements:

We thank Drew McDermott and Richard Stallman for suggestions, ideas and comments used in this paper. Jon Doyle is supported by a Fannie and John Hertz Foundation graduate fellowship. Guy Steele is supported by a National Science Foundation graduate fellowship. This research was conducted at the Artificial Intelligence Laboratory of the Massachusetts Institute of Technology. Support for the Laboratory's artificial intelligence research is provided in part by the Advanced Research Projects Agency of the Department of Defense under Office of Naval Research contract number N00014-75-C-0643.

Contents:

Section 1: The AMORD Reference Manual

AMORD[AMORD] is a problem solving system. AMORD encourages a style of expression in which the logical relationships of the knowledge and control structure of the problem solver are made explicit. A minimal set of mechanisms is supplied by AMORD so that most of the knowledge that must be formalized and the decisions that must be made in constructing a problem solving program must, to a large degree, be made explicit in AMORD. This makes AMORD is a vehicle for expressing the structure of problem solvers, such that once the problem solving structure has been formalized, the task of transferral to programs in programming languages is straightforward.

As a set of mechanisms, AMORD is a system for the pattern-directed invocation of a set of rules operating on an indexed data base of assertions. AMORD features a simple syntax for rule invocation patterns, an unconstrained format for assertions, unification semantics for the pattern-matcher, a non-chronological control structure for rule invocations, and the use of a truth maintenance system[TMS] for determining the current set of believed assertions. AMORD is implemented in SCHEME[SCHEME] implemented in MacLISP.[MacLISP]

The main components of AMORD are the data bases of assertions and rules, the TMS, the matcher, and the queue. The data bases used in storing assertions and rules are discrimination networks. The TMS is a system for maintaining the logical grounds for belief in assertions. The matcher is a syntactic unifier which has no distinguished positions or keywords. The queue is a system whereby rules are run on the appropriate assertions. The main loop of the AMORD interpreter is to simply run the body of all rules on all assertions whose patterns match the rules' patterns. This is done independent of the chronological order in which the assertions and rules are entered into the data bases. When all rules have been run on all matching facts, AMORD halts, awaiting further user input.

There are several special constructs in AMORD for expressing rules and assertions. We will enumerate them here, accompanied by their syntax and description.

ASSERT -- (ASSERT <pattern> <justification>)

This is the method for adding a new assertion to the data base. Any variables in the arguments inherit their values from the lexically surrounding text. Variables are denoted by atoms with a colon prefix, as in ":f". Each fact in the data base has an atomic factname. Assertions which are variants of each other designate the same fact in the data base, that is, are mapped to the same factname. The justification is a list, whose type is determined by the first element of the list. If the first element is atomic and has a "proof-type" function associated with it, that function is applied to the justification and assertion to construct the desired TMS justification. Otherwise, belief in the assertion is justified by belief in all of the facts in the rest of the justification.

Assertions are run on all matching rules.

RULE -- (RULE (<factname-variable> <pattern>) <body>)
This is the method for specifying a procedure to be invoked by all assertions matching <pattern>. When a fact whose pattern unifies with the rule pattern is inserted into the data base, the set of AMORD and SCHEME forms specified in the body of the rule are evaluated in the environment specified by adding {1} the variable bindings derived from the unification of the fact pattern and rule pattern to {2} the binding of the fact's factname and the factname variable of the rule pattern and {3} the bindings derived from the lexically surrounding text.[Godel] The primary use of the factname variable is for use in specifying justifications in assertions made in the rule body. Rules are run on all matching facts. The order in which they are run is not specified, although the interpreter of Section 4 operates in a quasi-depth-first fashion.

ASSUME -- (ASSUME <pattern> <justification>)
This is used to specify speculative hypotheses, that is, to assume a truth "for the sake of argument". Here the justification provides support for the need for assuming the assertion specified by pattern. Assumptions are made by justifying belief in the assumed assertion on the basis of a lack of belief in the assumed assertion's negation. Thus, assumptions may be discarded by justifying belief in the negation of the assumed assertion, which will invalidate the validity of the previous justification for this assumed fact. In particular, the dependency-directed backtracking mechanism of the TMS uses the information gained through analysis of the reasons for contradictions to retract conflicting assumptions in this manner.

The following macros can be used to interface expressions manipulated by the AMORD and SCHEME interpreters.

PDSVAL -- (PDSVAL <form>)
This macro allows SCHEME code to access the AMORD value of <form>.

PDSLET -- (PDSLET ((<var1> <val1>) ... (<varn> <valn>)) <body>)
This macro enables the binding of a number of AMORD variables to values expressed by SCHEME expressions. Note that the AMORD variables must be prefixed by a colon.

PDSCLOSE -- (PDSCLOSE <body>)
This macro allows the evaluation of AMORD forms from within SCHEME when the SCHEME expression being evaluated is not lexically surrounded by an AMORD expression. The forms in the body are evaluated in an empty AMORD environment, that is, an environment in which no AMORD variables are bound.

CONSTANT -- (CONSTANT <object>)
This SCHEME predicate determines whether an object contains any references to AMORD variables.

The following are used to initialize and invoke the AMORD interpreter.

INIT -- (INIT)
This function initializes the data bases and various system variables.

RUN -- (RUN)
This function initiates the AMORD read-evaluate loop. Forms read in this loop are closed in the empty environment and then evaluated. Unlike the SCHEME read-evaluate-print loop, the results of the evaluation of forms in this loop are not printed.

The following functions display the reasoning behind believed assertions.

WHY -- (WHY <factname>)
This prints the current justification for belief in the specified fact.

EXPLAIN -- (EXPLAIN <factname>)
This prints the complete proof of belief in the specified fact.

There are also a number of functions internal to the interpreter which are useful in writing specialized functions. The TMS functions and their use are described in Section 3. The most important are the following.

ASSERTION -- (ASSERTION <pattern>)
This returns the factname of the fact with the designated pattern.

FACT-STATEMENT -- (FACT-STATEMENT <factname>)
This returns the pattern associated with the designated fact.

RETRACT -- (RETRACT <factname>)
This removes all PREMISE type justifications possessed by the supplied fact.

There are several standard forms of justifications.

PREMISE -- (PREMISE)
This justification supports belief independent of any other beliefs.

GIVEN -- (GIVEN)
A synonym for PREMISE.

CONDITIONAL-PROOF -- (CONDITIONAL-PROOF <consequent> <hypotheses>)
This justification provides support if the recorded justifications

provide for belief in the consequent when all the hypotheses are believed. Actually, this justification type has a somewhat more complex capability and syntax which consistently extend the syntax and function just described. The concepts involved in this extension are described in Section 3, and the syntax is described in the annotated implementation in Section 4.

CP -- (CP <consequent> <hypotheses>)
A synonym for CONDITIONAL-PROOF.

CONTRADICTION -- (CONTRADICTION <support>)
This justification supports belief if all the facts in the mentioned support are believed, and further declares the fact justified by this justification to be a contradiction. This declaration will cause backtracking to be invoked whenever the justified fact is believed. All contradictions must be explicitly declared. That is, asserting facts which are syntactically negations of each other does not automatically produce a contradiction.

ASSUMPTION -- (ASSUMPTION <reason> <negation>)
This justification supports an assumed belief if the reason for making the assumption is believed and if there is no reason for believing the negation of the assumed fact. The negation used in this justification does not have to be a fact with a certain pattern, but merely any fact which will be taken as meaning (or at least implying) the negation of the assumed belief.

In addition to the above justification types, the justification types INSTANCE and RULE are used internally by the interpreter in making justifications based on subsumption of one fact by another and in justifying rules. These justification types should therefore be avoided by the user.

To use AMORD, simply incant at DDT (on MIT-AI):

```
:AMORD;AMORD
```

which will load up the current version of AMORD and enter the SCHEME read-evaluate-print loop. To enter the AMORD read-evaluate loop, evaluate the form (RUN), which will begin interpretation. To escape to LISP, type ^G. To restart SCHEME, type either ^^ or (SCHEME), from whence (RUN) can again be invoked to resume AMORD.

This concludes the AMORD reference manual.

Section 2: Some AMORD Examples

The control structure of AMORD encourages a certain style of rule-writing. In order to compute anything, the control of the computational process must be made explicit.[Explicit Control] The use of explicit control requires careful thought about making assertions with the correct justifications for belief. This section presents a simple system for deduction to illustrate these points.

The forward version of conjunction introduction is implemented in AMORD as the following rule:

```
(Rule (:f :a)
     (Rule (:g :b)
          (Assert (AND :a :b) (&+ :f :g))))
```

To paraphrase this rule, the addition of a fact F with pattern A into the data-base results in the addition of a rule which checks every fact G in the data-base and asserts the conjunction of A and the pattern B of G. Thus if A is asserted, so will be (AND A A), (AND A (AND A A)), (AND (AND A A) A), etc. Note that the atom AND is not a distinguished symbol.

Unfortunately, this rule is useless, as it generates piles of useless assertions. To control these deductions, the above rule can be replaced by the following rule which effects consequent reasoning about conjunctive goals.

```
(Rule (:g (SHOW (AND :p :q)))
     (Rule (:c1 :p)
          (Rule (:c2 :q)
               (Assert (AND :p :q) (&+ :c1 :c2)))
          (Assert (SHOW :q) ((BC &+) :g :c1)))
     (Assert (SHOW :p) ((BC &+) :g)))
```

In this rule the control statements (SHOWS) depend on belief in the relevant controlled facts so that the existence of a subgoal for the second conjunct of a conjunctive goal depends on the solution for the first conjunct. At the same time, no controlled facts depend on control facts, since the justification for a conjunction is entirely in terms of the conjuncts, and not on the need for deriving the conjunction. This means that the control over the derivation of facts cannot affect the truth of the derived facts. Moreover, the hierarchy of nested, lexically scoped rules allows the specification of sequencing and restriction information. For instance, the above rule could have been written as

```
(Rule (:g (SHOW (AND :p :q)))
      (Rule (:c1 :p)
            (Rule (:c2 :q)
                  (Assert (AND :p :q) (&+ :c1 :c2))))
      (Assert (SHOW :p) ((BC &+) :g))
      (Assert (SHOW :q) ((BC &+) :g)))
```

This form of the rule would also only derive correct statements, but would not be as tightly controlled as the previous rule. In this case, both subgoals are asserted immediately, although there is no reason to work on the second conjunct unless the first conjunct has been solved. This form of the rule allows more work to be done in that the possible mutual constraints of the conjuncts on each other due to shared variables is not accounted for. That is, in the first form of the rule, solutions to the first conjunct were used to specialize the subgoals for the second conjunct, so that the constraints of the solutions to the first are accounted for in the second subgoal. In the second form of the rule much work might be done on solving each subgoal independently, with the derivation of the conjunction performed by an explicit matching of these derived results. This allows solutions to the second subgoal to be derived which cannot match any solution to the first subgoal.

Other consequent rules for Modus Ponens, Negated Conjunction Introduction, and Double Negation Introduction are similar in spirit to the rule for Conjunction Introduction:

```
(Rule (:g (SHOW :q))
      (Rule (:i (-> :p :q))
            (Rule (:f :p)
                  (Assert :q (MP :i :f)))
            (Assert (SHOW :p) ((BC MP) :g :i))))

(Rule (:g (SHOW (NOT (AND :p :q))))
      (Rule (:t (NOT :p))
            (Assert (NOT (AND :p :q)) (-&+ :t)))
      (Rule (:t (NOT :q))
            (Assert (NOT (AND :p :q)) (-&+ :t)))
      (Assert (SHOW (NOT :p)) ((BC -&+) :g))
      (Assert (SHOW (NOT :q)) ((BC -&+) :g)))

(Rule (:g (SHOW (NOT (NOT :p))))
      (Rule (:f :p)
            (Assert (NOT (NOT :p)) (--+ :f)))
      (Assert (SHOW :p) ((BC --+) :g)))
```

The following two rules implement a consequent oracle for testing the equality of constants. Note the use of PDSVAL in allowing SCHEME access to the value of AMORD variables.

```
(Rule (:q (SHOW (= :a :b)))
      (let ((a (pdsval :a))
            (b (pdsval :b)))
           (if (constant a)
               (if (constant b)
                   (if (equal a b)
                       (Assert (= :a :b) (Equality)))))))

(Rule (:q (SHOW (NOT (= :a :b))))
      (let ((a (pdsval :a))
            (b (pdsval :b)))
           (if (constant a)
               (if (constant b)
                   (if (equal a b)
                       nil
                       (Assert (NOT (= :a :b)) (Equality)))))))
```

A final example is the use of assumptions to implement a default series of alternative choices. The following expresses the knowledge that traffic signals are either red, yellow or green.

```
(Rule (:t (TYPE :1 TRAFFIC-SIGNAL))
      (Assume (COLOR :1 GREEN) (Optimism :t))
      (Rule (:ng (NOT (COLOR :1 GREEN)))
            (Assume (COLOR :1 YELLOW) (Hope-Yet :t :ng))
            (Rule (:ny (NOT (COLOR :1 YELLOW)))
                  (Assert (COLOR :1 RED) (Rats :t :ng :ny)))))
```

By using this rule, anything declared to be a traffic signal will be assumed to be green in color. If it is discovered (perhaps due to a contradiction) that the color is not green, the color will be assumed to be yellow. If it is further discovered that the color is also not yellow, the color is determined to be red.

Section 3: The Use of the TMS in AMORD

The Truth Maintenance System is an independent program for recording information about program deductions. The TMS uses a method for representing knowledge about beliefs, called a non-monotonic dependency system, to effect any updating of beliefs necessary upon the addition of new information.

The basic operation of the TMS is to attach a justification to a TMS-node. A TMS-node can be linked with any component of program knowledge which is to be connected with other components of program information. In AMORD, each fact and rule has an associated TMS-node. The TMS then decides, on the basis of the justifications attached to nodes, which beliefs in the truth of nodes are supported by the recorded justifications. A node is said to be *in* if there is an associated justification which supports belief in the node. Otherwise, the node is said to be *out*. The TMS informs AMORD whenever the belief status of a node changes, either from *in* to *out*, or *out* to *in*.

There are several types of justifications supported by the TMS. The basic form of a justification is one in which a node is justified if each node in a set of other nodes is *in*. This type of justification represents the typical form of a deduction, or in the special case in which the set of other nodes is empty, a premise. A node may also be justified on the basis of the conditional proof of one node relative to a set of other nodes. In this, belief in the justified node is supported if the consequent node of the conditional proof is *in* when each of the nodes in the set of hypotheses is *in*. The remaining form of justification supports belief in a node if each node in a given set of other nodes is *out*. This non-monotonic justification allows the consistent representation and maintenance of hypothetical assumptions. Using this latter form of justification, a fact can be assumed to be true by justifying it on the basis of its negation being *out*.

Each node which is *in* has a distinguished element of its set of justifications. This distinguished justification is selected to support belief in the node in terms of other nodes having well-founded support, that is, non-circular proofs from ground hypotheses. A number of dependency relations are determined from these justifications, such as the set of nodes depending on a given node, or the nodes upon which a particular node depends.

Truth maintenance processing is required when new justifications cause changes in previously existing beliefs. In such cases, the status of all nodes depending on the nodes with changed beliefs must be redetermined. The critical aspect of this processing is ensuring that all nodes judged to be *in* are associated with well-founded support. Truth maintenance is not unlike a generalized, but incremental, form of garbage collection. The first step is to mark and collect all facts whose current belief state

depends, via the previously recorded consequence dependencies, on the changed beliefs. The second step is a combination sweep and depth first search over these facts with the purpose of determining belief states based on other facts with well-founded support. By distinguishing facts with well-founded support from those without, all new beliefs determined in this pass are guaranteed to be well-founded. The third step is necessary if the second step does not determine belief states for all the involved facts. This step consists of a relaxation process of assuming some belief states and proceeding, taking care that the assumed beliefs are consistent. This step, at its conclusion, can guarantee that all beliefs have well-founded support. The fourth step is a pass over all changed facts to check for believed facts which are known to represent contradictions. Backtracking is invoked on any such contradictions (which may so invoke further truth maintenance). The final step of truth maintenance is the notification of the external systems of all changes in beliefs determined by the truth maintenance system.

The TMS provides automatic dependency-directed backtracking whenever nodes marked as contradictions are brought *in*. Dependency-directed backtracking employs the recorded dependencies to locate precisely those hypotheses relevant to the failure and uses the conditional proof mechanism to summarize the cause of the contradiction in terms of these hypotheses. Because the reasons for the failure are summarized in a form which is independent of the hypotheses causing the failure, future occurrences of similar failures are avoided.

The TMS functions used in AMORD are as follows:

TMS-MAKE-DEPENDENCY-NODE -- (TMS-MAKE-DEPENDENCY-NODE <external-name>)
This function creates a new TMS-node with a given name. In AMORD, the external names are just the atomic factnames used to represent facts and rules. TMS-nodes are currently implemented using uninterned atomic symbols.

TMS-JUSTIFY -- (TMS-JUSTIFY <node> <insupporters> <outsupporters> <argument>)
This function gives a TMS node a new justification, which is valid if each of the nodes of the *in*supporters list is *in*, and each of the nodes of the *out*supporters list is *out*. The argument is an uninterpreted slot used to record the external form of the justification, and is retrievable via the TMS-ANTECEDENT-ARGUMENT function described below.

TMS-CP-JUSTIFY
-- (TMS-CP-JUSTIFY <node> <consequent> <inhypotheses> <outhypotheses> <argument>)
This gives a TMS node a new justification which is valid if, when the *in*hypotheses are *in* and the *out* hypotheses are *out*, the consequent node is believed. As in TMS-JUSTIFY, the argument is an uninterpreted record of the external form of the justification.

TMS-PROCESS-CONTRADICTION
-- (TMS-PROCESS-CONTRADICTION <name> <node> <type> <contradiction-function>)
This declares a TMS node to represent a contradiction. The name and type are uninterpreted mnemonics provided by the external system to be printed out during backtracking. The contradiction-function, if supplied, should be a LISP function to be called with the contradiction node as its argument when the backtracker can find no backtrackable choicepoints.

TMS-SUPPORT-STATUS -- (TMS-SUPPORT-STATUS <node>)
This function returns the support-status, either 'IN or 'OUT, of a node.

TMS-ANTECEDENT-SET -- (TMS-ANTECEDENT-SET <node>)
This function returns the list of justifications of the node. In the TMS, each justification is called an antecedent of the node.

TMS-SUPPORTING-ANTECEDENT -- (TMS-SUPPORTING-ANTECEDENT <node>)
This function returns the current justification of the node.

TMS-ANTECEDENT-ARGUMENT -- (TMS-ANTECEDENT-ARGUMENT <antecedent>)
This function returns the external argument associated with the given antecedent.

TMS-ANTECEDENTS -- (TMS-ANTECEDENTS <node>)
This function returns the list of nodes determining well-founded support for the given node. This list is extracted from the supporting-antecedent if the node is *in*, and is empty if the node is *out*.

TMS-CONSEQUENCES -- (TMS-CONSEQUENCES <node>)
This function returns the list of nodes whose list of antecedent nodes mentions the given node.

TMS-EXTERNAL-NAME -- (TMS-EXTERNAL-NAME <node>)
This function returns the user-supplied name of a node.

TMS-IS-IN -- (TMS-IS-IN <node>)
This predicate is true iff the node is *in*.

TMS-IS-OUT -- (TMS-IS-OUT <node>)
This predicate is true iff the node is *out*.

TMS-RETRACT -- (TMS-RETRACT <node>)
This function will remove all premise-type justifications from the set of justifications of the node.

TMS-PREMISES -- (TMS-PREMISES <node>)
This function returns a list of the premises among the well-founded support of the node.

TMS-ASSUMPTIONS -- (TMS-ASSUMPTIONS <node>)
This function returns a list of the assumptions among the well-founded support of the node.

TMS-CLOBBER-SIGNAL-FORGETTING-FUNCTION
-- (TMS-CLOBBER-SIGNAL-FORGETTING-FUNCTION <node> <fun>)
This function sets the LISP function that the TMS will use to signal the changing of the status of the node from *in* to *out*. When such a change occurs, the supplied function will be called with the external name of the node as its argument.

TMS-CLOBBER-SIGNAL-RECALLING-FUNCTION
-- (TMS-CLOBBER-SIGNAL-RECALLING-FUNCTION <node> <fun>)
This function sets the LISP function that the TMS will call with the node's external name as its argument when changing the status of the node from *out* to *in*.

The TMS also generates new "facts" internally during backtracking. These will therefore occur in explanations and antecedents of the nodes requested and justified by the external systems. The internal facts generated by the TMS are atoms with certain properties. The following functions are provided to manipulate these internal facts.

TMS-FACTP -- (TMS-FACTP <thing>)
This predicate is true iff the thing is an internal TMS fact.

TMS-FACT-NODE -- (TMS-FACT-NODE <fact>)
This function returns the TMS node associated with an internal fact.

TMS-FACT-STATEMENT -- (TMS-FACT-STATEMENT <fact>)
This function returns the symbolic statement of the meaning of an internal fact. This statement refers to the external names of the other facts, such as contradictions and assumptions, which were involved in the making of the fact.

The following two functions are supplied for debugging purposes.

TMS-INIT -- (TMS-INIT)
This function clears the state of the TMS by resetting all internal variables and clearing all properties and internings of TMS nodes.

TMS-INTERN -- (TMS-INTERN)
This function interns all TMS nodes currently in existence, and causes the interning of all nodes generated in the future. Initially, the atomic symbols representing TMS nodes are not interned.

Examples of the use of the TMS facilities can be found in the following section, in which the functions implementing the various AMORD proof-types are defined.

Section 4: An Annotated Interpreter

Here we present a real live AMORD interpreter. The interpreter divides into the following sections, which will be presented in this order.

AMORD form definitions
    ASSERT and associated functions
    RULE and associated functions
Proof-type definitions
The RUN interpreter (the main loop)
The TMS interface
The Unification Matcher
The Discrimination-Net Data Base

Before presenting the interpreter itself, we describe some aspects of the implementation.

The main loop of the interpreter is in the function RUN, which examines the various queues (described below). RUN makes sure that all rules are run on all facts whose patterns match the rule patterns. As an efficiency step, a rule is run on a fact only if both the rule and fact are believed (*in*). After the possibilities for running rules on facts are exhausted, RUN checks for programs (called runlast functions) which have been specified for running at queue's end and runs each of these programs. If these programs make new assertions or rules, the above loop is resumed. Finally, after finishing all of the above steps, RUN waits for new input from the user.

Each rule and fact is represented by an atomic symbol with several properties. Both rules and facts have their TMS-nodes kept on their property-list under the 'TMS-NODE property. Rules and facts also have a 'STIMULATE-LIST property, which is used to store matching facts and rules (respectively) until they are queued up to be run.

In addition to their common properties, rules and facts have other attached items. Facts have their pattern kept in their value cell. Rules have their full trigger pattern (the list of the factname variable and the trigger pattern proper) kept in their value cell. Rules are distinguished from facts by their possession of a 'RULE-BODY property, which stores the uninstantiated rule body. Rules also have a 'SPECIALIZATION property which stores the environment derived from the lexically surrounding text, and a 'T-LIST property, which stores the lexically surrounding triggering facts (the list of facts triggering lexically surrounding rules to create the particular rule).

The control of running rules on facts is mediated by an amorphous mechanism called the queue. This mechanism has several components:

(1) The trigger queue, *TQ*. This is a queue of rule-fact pairs representing possible triggerings. This queue is maintained, in the global

variable *TQ*, as a CONS cell, the CAR of which points to the front of the list of trigger pairs, and the CDR of which points to the last cell of this list. This is done so that new pairs may be quickly added to the end of the list of trigger pairs. The rule-fact pairs from this queue are turned into SCHEME closures and then run. The actual unification checking (over the matching done by the data base fetch routines) to see that the triggering is valid is done at closure creation time.

{2} The stimulate lists. Each rule and fact has a list, of facts and rules respectively, on its 'STIMULATE-LIST property. These facts and rules in these lists are initially the items retrieved from the data base as possibly matching the newly created rule or fact. The function STIMULATE, called by the TMS when rules and facts come *in*, takes the STIMULATE-LIST of the newly *in*ned item, turns it into a list of pairs and adds these pairs to the trigger queue.

The queue mechanism operates as follows. When pairs come to the top of the trigger queue, both the rule and the fact of the pair are checked to see if they are *in*. If both are *in*, their unification is attempted. If they do not unify, the pair is discarded from the queueing system: if they do, a SCHEME closure of the appropriate form is created and evaluated. This closure evaluates each form in the rule body using the inherited AMORD lexical environment augmented by the bindings derived from the triggering fact. Alternatively, if a pair is encountered on the trigger queue with the rule (or fact) *out*, the fact (or rule) is placed on the STIMULATE-LIST of the *out* rule (or fact). In this way no pairs are actually run unless relevant, for subsequent *in*nings of the rules or facts involved will keep adding the pair to the trigger queue until the pair makes it to the top with both items *in*.

In addition to the above trigger queue mechanism, two other structures are part of the main RUN loop.

{1} The closure queue, *Q*. This is queue of SCHEME closures, functions of no arguments to be evaluated. The global variable *Q* contains this queue, in the form of a CONS whose CAR is the first cell of the list forming the queue, and whose CDR is the last cell of this list. As in the trigger queue, this is done so that new queue items can be added directly at the end of the queue, rather than requiring a traversal through the entire queue for each new addition. This queue is provided so that the user may post programs to be executed. This is sometimes (although rarely) necessary, as the TMS makes the restriction that the TMS cannot be invoked while a previous invocation is still signalling changes in the statuses of facts.

{2} The runlast list, *RUNLAST*. This is a user maintained list, initially empty, of SCHEME functions of no arguments to be run each time both *TQ* and *Q* run out. At such time, each function in this list is evaluated. These functions can either add new justifications to facts, add other programs to *Q* to be run, or, by means of PDSCLOSE, evaluate further AMORD forms to cause resumption of the main loop of trigger queue interpretation.

The structure of justifications is as follows. Justifications must be lists. If the first element of the list is either non-atomic, or lacks a 'PROOF-TYPE property if atomic, the justification is interpreted as a simple deductive justification in which the justified item will be *in* if all the facts mentioned in the rest of the justification are *in*. If the first element of the justification is an atom with a 'PROOF-TYPE property, the the value of that property must be a SCHEME function. This function is called with the justification and justified item as arguments. This function then has the responsibility for making the necessary TMS justifications, and may perform other operations if desired. Proof-type functions which must evaluate AMORD forms should use the PDSCLOSE macro described in Section 1.

The interpreter uses several global variables as follows:

Q - The queue containing SCHEME closures to run.
TQ - The trigger queue containing rule-fact pairs to close and run.
ENTRY - Contains the last closure evaluated by RUN.
RUNLAST - A list of SCHEME closures of no arguments to be successively evaluated each time the queue runs out. This list is initially NIL.
STOPFLAG - If non-NIL, causes the RUN loop to halt after running the current entry.
ASSERTIONS - Contains the discrimination net for facts.
RULES - Contains the discrimination net for rules.
WALLP - If non-NIL, causes new justifications of facts to be displayed. The default is T.
GENSYM-COUNTER - The counter used in generating rule and fact names, numbers for standardizing expressions apart, and line numbers.

Here begins the code of the interpreter proper. Several macros are used in this code, including the substituting-quote ", which returns the next form, quoted but with the values of subforms preceded by , substituted as elements of list structure, and with the values of subforms preceded by @ spliced in as list segments. The macros IF and LET have the obvious meanings.

The first items are declarations for the SCHEME[RABBIT] and MacLISP compilers, respectively.

```
; RABBIT COMPILER DECLARATIONS
(PROCLAIM (*EXPR GENS ENQUEUE FACT-STATEMENT RULE-PATTERN SUPPORT-STATUS IS-IN
                 TMS-CLOBBER-SIGNAL-RECALLING-FUNCTION
                 TMS-MAKE-DEPENDENCY-NODE TMS-NODE TMS-NODES
                 TMS-JUSTIFY TMS-CP-JUSTIFY TMS-PROCESS-CONTRADICTION))

(DECLARE (FASLOAD (GJS) SCHMAC)) ;LOADS IN IF, ETC. MACROS FOR USE IN LISP
```

AMORD FORM DEFINITIONS

All true AMORD forms like ASSERT and RULE must be evaluated in a SCHEME environment in which the variables *SUBSTITUTION* and *T-LIST* are bound. To achieve this, while making these universal (not global) variables invisible to the user, macros are used which append the appropriate variable references to the calls to the AMORD primitives.

Here is ASSERT, which takes an expression and a justification, instantiates them with the current environment bindings, inserts the expression into the data base, and then installs the justification as one of the expression's justifications. The call to SUBSUME-CHECK serves to add new justifications to the new fact or to other facts based on subsumptions in their patterns.

```
(SCHMAC ASSERT (EXPRESSION JUSTIFICATION)
       "(ASSERT-2 ',EXPRESSION ',JUSTIFICATION *SUBSTITUTION*))

(DEFINE ASSERT-2
   (LAMBDA (EXPRESSION JUSTIFICATION ALIST)
         (LET ((EXPRESSION (INSTANCE EXPRESSION ALIST))
               (JUSTIFICATION (INSTANCE JUSTIFICATION ALIST)))
              (LET ((A (ASSERTION EXPRESSION)))
                   (BLOCK (INSTALL-JUST JUSTIFICATION A)
                          (SUBSUME-CHECK A))))))
```

The operation of ASSUME is somewhat more complicated than that of ASSERT, as two facts are created in addition to the specified fact, as well as one additional justification.

```
(SCHMAC ASSUME (EXPRESSION JUSTIFICATION)
       "(ASSUME-2 ',EXPRESSION ',JUSTIFICATION *SUBSTITUTION*))

(DEFINE ASSUME-2
   (LAMBDA (EXPRESSION JUSTIFICATION ALIST)
         (LET ((EXPRESSION (INSTANCE EXPRESSION ALIST))
               (JUSTIFICATION (INSTANCE JUSTIFICATION ALIST)))
              (LET ((A (ASSERTION EXPRESSION))
                    (AF (ASSERTION "(ASSUMED ,EXPRESSION)))
                    (N (ASSERTION
                        (IF (EQ (CAR EXPRESSION) 'NOT)
                            (CADR EXPRESSION)
                            "(NOT ,EXPRESSION)))))
                   (BLOCK (INSTALL-JUST JUSTIFICATION AF)
                          (INSTALL-JUST "(ASSUMPTION ,AF ,N) A)
                          (SUBSUME-CHECK A)
                          (SUBSUME-CHECK AF)
                          (SUBSUME-CHECK N))))))
```

ASSERTION is the function for creating new assertions. The data base

is checked to see if it contains a fact with a variant of the supplied pattern. If so, that fact is returned, and otherwise a new fact is generated and inserted into the data base in the appropriate bucket.

```
(DEFINE ASSERTION
   (LAMBDA (EXPRESSION)
      (LET ((B (BUCKET EXPRESSION NIL *ASSERTIONS*)))
           (VARIANT-CHECK EXPRESSION B
            (LAMBDA (VARIANT) VARIANT)          ;IF THERE IS A VARIANT
            (LAMBDA ()                          ;NO VARIANT
                    (LET ((NAME (GENS 'F)))
                         (BLOCK (SET NAME EXPRESSION)
                                (PUTPROP NAME (TMS-MAKE-DEPENDENCY-NODE NAME) 'TMS-NODE)
                                (TMS-CLOBBER-SIGNAL-RECALLING-FUNCTION
                                 (TMS-NODE NAME) 'STIMULATE)
                                (PUTPROP NAME
                                         (DO ((L (FETCH EXPRESSION NIL *RULES*) ((CADR L)))
                                              (ANS NIL (CONS (CAR L) ANS)))
                                             ((NULL L) ANS))
                                         'STIMULATE-LIST)
                                (INSERT-IN-BUCKET NAME B)
                                NAME)))))))
```

VARIANT-CHECK is a function used only by ASSERTION above. It checks a data base bucket to see if the bucket contains a fact whose pattern is a variant of the supplied pattern. IF-FOUND should be a function of one argument to receive the variant if one is found. IF-NOT should be a function of no arguments to be called if no variant is found.

```
(DEFINE VARIANT-CHECK
   (LAMBDA (EXP BUCKET IF-FOUND IF-NOT)
      (LABELS ((LOOP
                (LAMBDA (L)
                   (IF L
                       (LET ((C (COMPARE EXP (FACT-STATEMENT (CAR L)))))
                            (IF C
                                (IF (EQ (CAR C) 'VARIANT)
                                    (IF-FOUND (CAR L))
                                    (LOOP (CDR L)))
                                (LOOP (CDR L))))
                       (IF-NOT)))))
        (LOOP (STUFF BUCKET)))))
```

SUBSUME-CHECK performs the function of checking the data base for facts whose patterns either subsume or are subsumed by the pattern of the supplied fact. If any subsumptions are detected, new justifications are added to support belief in the subsumed fact if the subsuming fact is believed.

```
(DEFINE SUBSUME-CHECK
   (LAMBDA (NAME)
      (LET ((EXP (FACT-STATEMENT NAME)))
           (DO ((CANDIDATES (FETCH EXP NIL *ASSERTIONS*) ((CDDR CANDIDATES))))
               ((NULL CANDIDATES))
               (IF (EQ (CAR CANDIDATES) NAME)
                   NIL
                   (LET ((C (COMPARE EXP (FACT-STATEMENT (CAR CANDIDATES)))))
                        (IF C
                            (COND ((EQ (CAR C) 'SUBSUMES)
                                   (INSTALL-JUST (LIST 'INSTANCE NAME) (CAR CANDIDATES)))
                                  ((EQ (CAR C) 'SUBSUMED)
                                   (INSTALL-JUST (LIST 'INSTANCE (CAR CANDIDATES)) NAME))
                                  (T (BREAK |SUBSUME-CHECK|)))))))))
```

The next function is not used in the interpreter, but provides a useful service in writing AMORD rules and proof types. PRESENT takes two arguments -- a full rule pattern of the form (<factname> <pattern>) and IF-FOUND, a continuation of two arguments. If a fact is found which is subsumed by the pattern, IF-FOUND is called with the resulting substitution and a continuation of no arguments which can be called to continue the scan. To use the derived substitution in the evaluation of AMORD forms, the continuation IF-FOUND should use "*SUBSTITUTION*" as the name of its first argument.

```
(DEFINE PRESENT
   (LAMBDA (PATTERN IF-FOUND)
      (LABELS ((LOOP
                (LAMBDA (CANDIDATES)
                   (IF CANDIDATES
                       (LET ((C (COMPARE (CADR PATTERN) (FACT-STATEMENT (CAR CANDIDATES)))))
                            (IF C
                                (IF (EQ (CAR C) 'SUBSUMES)
                                    (IF-FOUND (CONS (CONS (CAR PATTERN) (CAR CANDIDATES))
                                                    (CADR C))
                                              (LAMBDA () (LOOP ((CDDR CANDIDATES)))))
                                    (LOOP ((CDDR CANDIDATES))))
                                (LOOP ((CDDR CANDIDATES)))))
                       NIL))))
        (LOOP (FETCH (CADR PATTERN) NIL *ASSERTIONS*)))))
```

Rules have justifications just like facts, but unlike facts, rules are used in no justifications. Rules are really operational entities, which should be allowed to function only if the facts leading to their creation (via other rules forming its lexical environment) are believed. For this purpose, each rule has a 'T-LIST property storing the list of facts which triggered rules forming its lexical environment. This list, augmented with the rule itself, is passed along to nested rules by means of the variable *T-LIST*, a universal variable like *SUBSTITUTION*.

```
(SCHMAC RULE (PATTERN . BODY)
       "(RULE-2 ',PATTERN ',BODY *SUBSTITUTION* *T-LIST*))

(DEFINE RULE-2
   (LAMBDA (PATTERN BODY ALIST T-LIST)
     (IF (NULL BODY) (ERROR '|VACUOUS RULE| PATTERN 'WRNG-TYPE-ARG)
         (LET ((B (BUCKET (CADR PATTERN) ALIST *RULES*))
               (RNAME (GENS 'R)))
              (BLOCK (PUTPROP RNAME ALIST 'SPECIALIZATION)
                     (SET RNAME PATTERN)
                     (PUTPROP RNAME
                              (IF (CDR BODY) (CONS 'BLOCK BODY) (CAR BODY))
                              'RULE-BODY)
                     (PUTPROP RNAME T-LIST 'T-LIST)
                     (PUTPROP RNAME
                              (DO ((L (FETCH (CADR PATTERN) ALIST *ASSERTIONS*) ((CADR L)))
                                   (ANS NIL (CONS (CAR L) ANS)))
                                  ((NULL L) ANS))
                              'STIMULATE-LIST)
                     (INSERT-IN-BUCKET RNAME B)
                     (PUTPROP RNAME (TMS-MAKE-DEPENDENCY-NODE RNAME) 'TMS-NODE)
                     (TMS-CLOBBER-SIGNAL-RECALLING-FUNCTION (TMS-NODE RNAME) 'STIMULATE)
                     (INSTALL-JUST "(RULE @T-LIST) RNAME))))))
```

TRY-RULE takes a possible triggering pair, consisting of a rule and a fact. The pattern of the fact is compared with the pattern of the rule. If these two patterns unify, then a SCHEME function of no arguments is returned which, if evaluated, will evaluate the body of the rule in the environment produced by adding the bindings derived from the unification to the environment in which the rule exists.

```
(DEFINE TRY-RULE
   (LAMBDA (RNAME ANAME)
          (LET ((S (UNIFY (CADR (RULE-PATTERN RNAME))
                          (FACT-STATEMENT ANAME) (GET RNAME 'SPECIALIZATION))))
               (IF S
                   (ENCLOSE
                    "(LAMBDA ()
                             (LET ((*SUBSTITUTION* '((,(CAR (RULE-PATTERN RNAME)) . ,ANAME)
                                                     . ,(CAR S)))
                                   (*T-LIST* ',(CONS ANAME (GET RNAME 'T-LIST))))
                                  ,(GET RNAME 'RULE-BODY)))
                             RNAME)))))
```

PROOF-TYPES AND JUSTIFICATIONS

INSTALL-JUST takes a justification and a fact (or rule). If the justification has an associated proof-type, the proof-type function is called with the justification and fact as arguments. Otherwise, SUPPORT is

called to add the justification to the set of justifications of the fact. If the new justification causes the fact to be newly believed, the fact and its justification may be displayed.

```
(DEFINE INSTALL-JUST
   (LAMBDA (JUSTIFICATION FACT)
      (LET ((OLDSTATUS (SUPPORT-STATUS FACT)))
        (IF (ATOM (CAR JUSTIFICATION))
            (LET ((G (GET (CAR JUSTIFICATION) 'PROOF-TYPE)))
                 (IF G (G JUSTIFICATION FACT) (SUPPORT JUSTIFICATION FACT)))
            (SUPPORT JUSTIFICATION FACT))
        (IF (AND *WALLP*
                 (NULL (GET FACT 'RULE-BODY))   ;FACT OR RULE?
                 (EQ OLDSTATUS 'OUT)
                 (EQ (SUPPORT-STATUS FACT) 'IN))
            (BLOCK (PRINT 'ASSERTING)
                   (PRIN1 FACT)
                   (PRINC '| |)
                   (PRIN1 (FACT-STATEMENT FACT))
                   (PRINC '| |)
                   (PRIN1 JUSTIFICATION))))))

(SET' *WALLP* T)
```

SUPPORT performs the standard task of justification, which interprets all elements of the supplied justification (except the first, which is mnemonic) to be factnames which collectively justify belief in the supplied fact.

```
(DEFINE SUPPORT
        (LAMBDA (JUSTIFICATION FACT)
                (TMS-JUSTIFY (TMS-NODE FACT)
                             (TMS-NODES (CDR JUSTIFICATION))
                             NIL
                             JUSTIFICATION)))
```

PREMISE justifies the fact with a eternally valid justification.

```
(DEFINE PREMISE
        (LAMBDA (JUSTIFICATION FACT)
                (TMS-JUSTIFY (TMS-NODE FACT) NIL NIL JUSTIFICATION)))

(PUTPROP 'PREMISE PREMISE 'PROOF-TYPE)
(PUTPROP 'GIVEN PREMISE 'PROOF-TYPE)
```

CONDITIONAL-PROOF interprets the second element of the justification as the consequent of the conditional proof, the third element as the list of *in* hypotheses of the conditional proof, and the fourth element as the list of *out* hypotheses of the conditional proof.

```
(DEFINE CONDITIONAL-PROOF
        (LAMBDA (JUSTIFICATION FACT)
                (TMS-CP-JUSTIFY (TMS-NODE FACT)
                                (TMS-NODE (CADR JUSTIFICATION))
                                (TMS-NODES (CADDR JUSTIFICATION))
                                (TMS-NODES (CADDDR JUSTIFICATION))
                                JUSTIFICATION)))

(PUTPROP 'CP CONDITIONAL-PROOF 'PROOF-TYPE)
(PUTPROP 'CONDITIONAL-PROOF CONDITIONAL-PROOF 'PROOF-TYPE)
```

ASSUMPTION interprets the second element of the justification as a factname designating the reason for making the assumption, and the third element as a factname designating a negation of the belief to be assumed. Thus the supplied fact will be believed whenever the reason fact is *in*, and the negation fact is *out*.

```
(DEFINE ASSUMPTION
        (LAMBDA (JUSTIFICATION FACT)
                (TMS-JUSTIFY (TMS-NODE FACT)
                        (LIST (TMS-NODE (CADR JUSTIFICATION)))
                        (LIST (TMS-NODE (CADDR JUSTIFICATION)))
                        JUSTIFICATION)))

(PUTPROP 'ASSUMPTION ASSUMPTION 'PROOF-TYPE)
```

CONTRADICTION first supports belief in the supplied fact and then declares to the TMS that the fact is a contradiction.

```
(DEFINE CONTRADICTION
   (LAMBDA (JUST FACT)
       (BLOCK
        (SUPPORT JUST FACT)
        (TMS-PROCESS-CONTRADICTION FACT (TMS-NODE FACT) (FACT-STATEMENT FACT) NIL))))

(PUTPROP 'CONTRADICTION CONTRADICTION 'PROOF-TYPE)
```

THE RUN INTERPRETER

The following three macros hide references to the universal AMORD variables *SUBSTITUTION* and *T-LIST*, allowing SCHEME and AMORD code to be mixed.

```
(SCHMAC PDSVAL (ID) "(INSTANCE ',ID *SUBSTITUTION*))

(SCHMAC PDSLET (VARS . BODY)
       "(LET ((*SUBSTITUTION*
               ,(DO ((A '*SUBSTITUTION*
                        "(CONS (CONS ',(CAAR VL) ,(CADAR VL))
                               ,A))
                     (VL VARS (CDR VL)))
                    ((NULL VL) A))))
             @BODY))

(SCHMAC PDSCLOSE BODY "(LET ((*SUBSTITUTION* NIL) (*T-LIST* NIL)) @BODY))
```

RUN has four loops in one. First the trigger queue is tried, then the main queue, then the runlast functions, and finally the reader is invoked. The loop is halted if *STOPFLAG* is non-NIL.

```
(DEFINE RUN
  (LAMBDA ()
    (LABELS
      ((LOOP
        (LAMBDA () (IF *STOPFLAG* 'STOPPED (TRY-*TQ*))))
       (TRY-*TQ*
        (LAMBDA ()
            (IF (CAR *TQ*)
                (LET ((R (CAAAR *TQ*)) (F (CDAAR *TQ*)))
                     (BLOCK
                      (RPLACA *TQ* (CDAR *TQ*))
                      (IF (IS-IN F)
                          (IF (IS-IN R)
                              (BLOCK (SET' *ENTRY* (TRY-RULE R F)) (IF *ENTRY* (*ENTRY*)))
                              (PUTPROP R (CONS F (GET R 'STIMULATE-LIST)) 'STIMULATE-LIST))
                          (PUTPROP F (CONS R (GET F 'STIMULATE-LIST)) 'STIMULATE-LIST))
                      (LOOP)))
                (TRY-*Q*))))
       (TRY-*Q*
        (LAMBDA ()
            (IF (CAR *Q*)
                (BLOCK (SET' *ENTRY* (CAAR *Q*))
                       (RPLACA *Q* (CDAR *Q*))
                       (*ENTRY*)
                       (LOOP))
                (TRY-*RUNLAST*))))
       (TRY-*RUNLAST*
        (LAMBDA ()
            (DO ((RL *RUNLAST* (CDR RL)))
                ((NULL RL)
                 (IF (OR (CAR *TQ*) (CAR *Q*)) (LOOP) (TRY-READ)))
                ((CAR RL)))))
       (TRY-READ
        (LAMBDA ()
           (BLOCK
            (SET' *GENSYM-COUNTER* (+ *GENSYM-COUNTER* 1))
            (PRINT *GENSYM-COUNTER*)
            (PRINC '|>> |)
            (ENQUEUE (LIST (ENCLOSE `(LAMBDA () (PDSCLOSE ,(READ))) '?)))
            (LOOP)))))
      (BLOCK (SET' *STOPFLAG* NIL) (LOOP)))))
```

ENQUEUE adds a list of closures to the end of the current queue of closures.

```
(DEFUN ENQUEUE (ACTIONS)
       (IF ACTIONS
           (LET ((L (LAST ACTIONS)))
                (IF (CAR *Q*)
                    (PROGN (RPLACD (CDR *Q*) ACTIONS) (RPLACD *Q* L))
                    (PROGN (RPLACA *Q* ACTIONS) (RPLACD *Q* L))))))
```

STIMULATE is the (LISP) function called by the TMS on any fact or rule which changes status from *out* to *in*. (See ASSERTION and RULE-2 above for the uses of TMS-CLOBBER-SIGNAL-RECALLING-FUNCTION to implement this.) When such a status change takes place, the list of matching items found when the item was inserted into the data base is used to add a new set of trigger pairs to the trigger queue.

```
(DEFUN STIMULATE (NAME)
       (LET ((ACTIONS (IF (GET NAME 'RULE-BODY)
                          (MAPCAR '(LAMBDA (F) (CONS NAME F)) (GET NAME 'STIMULATE-LIST))
                          (MAPCAR '(LAMBDA (R) (CONS R NAME)) (GET NAME 'STIMULATE-LIST)))))
            (PROGN
             (REMPROP NAME 'STIMULATE-LIST)
             (IF ACTIONS
                 (LET ((L (LAST ACTIONS)))
                      (IF (CAR *TQ*)
                          (PROGN (RPLACD (CDR *TQ*) ACTIONS) (RPLACD *TQ* L))
                          (PROGN (RPLACA *TQ* ACTIONS) (RPLACD *TQ* L))))))))
```

INIT does the obvious thing.

```
(DEFINE INIT
        (LAMBDA ()
                (BLOCK (DBINIT '*ASSERTIONS*)
                       (DBINIT '*RULES*)
                       (SET' *Q* (CONS NIL NIL)) ;CAR IS FIRST CELL OF QUEUE, CDR IS LAST CELL
                       (SET' *TQ* (CONS NIL NIL))
                       (SET' *RUNLAST* NIL)
                       (SET' *ENTRY* NIL)
                       (SET' *STOPFLAG* NIL)
                       (SET' *GENSYM-COUNTER* 0))))
```

Variables are represented by semi-lists of three elements, in the form (/: <var> . <number>) The first element is the atom ":", the second is the variable name, and the third is a number used to standardize the variable name apart. The following functions should be used to create new variables and to test or otherwise manipulate them.

```
(DEFUN VGENS (VNAME)
       (CONS '/: (CONS (CAR VNAME)
                       (SET' *GENSYM-COUNTER* (+ *GENSYM-COUNTER* 1)))))

(DEFUN VARIABLE (X) (EQ (CAR X) '/:))
```

CONSTANT tests whether an S-expression contains any variables.

```
(DEFUN CONSTANT (X)
       (IF (ATOM X)
           (IF (EQ X '/:) NIL X)
           (IF (CONSTANT (CAR X)) (CONSTANT (CDR X)) NIL)))
```

GENS generates a new atomic symbol with a supplied prefix and a suffix of the form "-nnn".

```
(DEFUN GENS (E)
       (READLIST (NCONC (EXPLODE E)
                        (LIST '-)
                        (EXPLODE (SET' *GENSYM-COUNTER*
                                       (+ *GENSYM-COUNTER* 1))))))
```

The variable designator ":" is a read macro which generates the standard variable-structure described above. Because items read in see a constant value for *GENSYM-COUNTER*, variable references in an expression (such as two occurrences of ":x") appear as similar structures (such as "(/: x . 127)").

```
(DEFUN COLON-READ () (CONS '/: (CONS (READ) *GENSYM-COUNTER*)))

(SETSYNTAX '/: 'MACRO 'COLON-READ)
```

THE TMS INTERFACE

FACT-STATEMENT must check to see if the supplied fact is TMS-generated or a normal fact. RULE-PATTERN need make no such check.

```
(DEFUN FACT-STATEMENT (F) (IF (TMS-FACTP F) (TMS-FACT-STATEMENT F) (SYMEVAL F)))

(DEFUN RULE-PATTERN (R) (SYMEVAL R))
```

WHY presents the immediate justification for the current belief in a fact. Note that if the fact is not believed, the list of failing justifications is printed. EXPLAIN collects up all facts among the support of the supplied fact, sorts them by the suffix of their factname, and prints them one per line along with their current justifications.

```
(DEFUN WHY (NAME)
       (PRINT NAME)
       (PRIN1 (FACT-STATEMENT NAME))
       (PRINC '| |)
       (IF (IS-IN NAME)
           (PRIN1 (ARGUMENT NAME))
           (PRIN1 (CONS 'OUT
                        (MAPCAR 'ARGUMENT (ANTECEDENT-SET NAME)))))
       'QED)

(DEFUN EXPLAIN (FACT)
       (TERPRI) (PRINC '|PROOF OF |) (PRIN1 FACT) (PRINC '| = |) (PRIN1 (FACT-STATEMENT FACT))
       (PRINC '| (|) (PRIN1 (SUPPORT-STATUS FACT)) (PRINC '|) |) (PRIN1 (ARGUMENT FACT))
       (PFL (FOUNDATIONS FACT))
       'QED)
```

The following functions do the dirty work for functions like EXPLAIN.

```
(DEFUN PFL (FL)
       (MAPC '(LAMBDA (F)
                (PRINT F)
                (PRINC '| = |)
                (PRIN1 (FACT-STATEMENT F))
                (PRINC '| (|) (PRIN1 (SUPPORT-STATUS F)) (PRINC '|) |)
                (PRIN1 (ARGUMENT F)))
             (SORT (APPEND FL NIL) 'FACT-NAME-ALPHAGREATERP)))

(DEFUN FACT-NAME-ALPHAGREATERP (F G)
       (GREATERP (GENS-NUMBER-EXTRACT F) (GENS-NUMBER-EXTRACT G)))

(DEFUN GENS-NUMBER-EXTRACT (X)
       (DO ((E (CDR (MEMQ '- (EXPLODE X))) (CDR (MEMQ '- E))))
           ((NOT (MEMQ '- E)) (READLIST E))))
```

TMS-NODE returns the TMS node associated with a rule or fact. The error check is useful, in that a frequent mistake is to specify a justification with a constant in the support by forgetting to prefix a variable name with a colon.

```
(DEFUN TMS-NODE (F)
       (LET ((N (IF (TMS-FACTP F) (TMS-FACT-NODE F) (GET F 'TMS-NODE))))
            (IF N N (ERROR '|BAD ARGUMENT TO TMS-NODE| F 'WRNG-TYPE-ARG))))

(DEFUN TMS-NODES (L) (MAPCAR 'TMS-NODE L))
```

The following serve to interface the TMS to AMORD.

```
(DEFUN SUPPORT-STATUS (FACT) (TMS-SUPPORT-STATUS (TMS-NODE FACT)))

(DEFUN ARGUMENT (FACT) (TMS-ANTECEDENT-ARGUMENT (TMS-SUPPORTING-ANTECEDENT (TMS-NODE FACT))))
```

```
(DEFUN ANTECEDENT-SET (FACT) (TMS-ANTECEDENT-SET (TMS-NODE FACT)))

(DEFUN SUPPORTING-ANTECEDENT (FACT) (TMS-SUPPORTING-ANTECEDENT (TMS-NODE FACT)))

(DEFUN ANTECEDENTS (FACT)
       (MAPCAR 'TMS-EXTERNAL-NAME (TMS-ANTECEDENTS (TMS-NODE FACT))))

(DEFUN CONSEQUENCES (FACT)
       (MAPCAR 'TMS-EXTERNAL-NAME (TMS-CONSEQUENCES (TMS-NODE FACT))))

(DEFUN IS-IN (FACT) (TMS-IS-IN (TMS-NODE FACT)))

(DEFUN IS-OUT (FACT) (TMS-IS-OUT (TMS-NODE FACT)))

(DEFUN ARE-IN (FACTS) (TMS-ARE-IN (TMS-NODES FACTS)))

(DEFUN ARE-OUT (FACTS) (TMS-ARE-OUT (TMS-NODES FACTS)))

(DEFUN FOUNDATIONS (FACT)
       (MAPCAR 'TMS-EXTERNAL-NAME (TMS-ALL-ANTECEDENTS (TMS-NODE FACT))))

(DEFUN REPERCUSSIONS (FACT)
       (MAPCAR 'TMS-EXTERNAL-NAME (TMS-ALL-CONSEQUENCES (TMS-NODE FACT))))

(DEFUN PREMISES (NAME) (MAPCAR 'TMS-EXTERNAL-NAME (TMS-PREMISES (TMS-NODE NAME))))

(DEFUN ASSUMPTIONS (NAME) (MAPCAR 'TMS-EXTERNAL-NAME (TMS-ASSUMPTIONS (TMS-NODE NAME))))

(DEFUN RETRACT (NAME) (TMS-RETRACT (TMS-NODE NAME)))
```

THE UNIFICATION MATCHER

```
(PROCLAIM (*EXPR RASSOC VARIABLE VGENS))
```

COMPARE takes two expressions, A and B, as input. If B is a variant of A it returns (VARIANT <substitution>). If A subsumes B it returns (SUBSUMES <substitution>). If B subsumes A it returns (SUBSUMED <substitution>). Otherwise it returns NIL.

```
(DEFINE COMPARE
   (LAMBDA (A B)
      (LABELS ((MATCH
                (LAMBDA (A B S TYPE C)
                   (COND ((EQ A B) (C S TYPE))
                         ((AND (NUMBERP A) (NUMBERP B)) (IF (= A B) (C S TYPE)))
```

```
((EQ TYPE 'VARIANT)
 (COND ((ATOM A) (MATCH A B S 'SUBSUMED C))
       ((VARIABLE A)
        (IF (AND (NOT (ATOM B)) (VARIABLE B))
            (LET ((VCELL (ASSOC A S)))
                 (IF VCELL
                     (IF (EQUAL (CDR VCELL) B)
                         (C S 'VARIANT)
                         (MATCH A B S 'SUBSUMED C))
                     (IF (RASSOC B S)
                         (MATCH A B S 'SUBSUMES C)
                         (C (CONS (CONS A B) S)
                            'VARIANT))))
            (MATCH A B S 'SUBSUMES C)))
       ((ATOM B) NIL)
       ((VARIABLE B)
        (MATCH A B S 'SUBSUMED C))
       (T (MATCH (CAR A) (CAR B) S TYPE
            (LAMBDA (S TYPE)
               (MATCH (CDR A) (CDR B) S TYPE C))))))
((EQ TYPE 'SUBSUMES)
 (COND ((ATOM A) NIL)
       ((VARIABLE A)
        (LET ((VCELL (ASSOC A S)))
             (IF VCELL
                 (IF (EQUAL (CDR VCELL) B)
                     (C S TYPE)
                     NIL)
                 (C (CONS (CONS A B) S) TYPE))))
       ((ATOM B) NIL)
       (T (MATCH (CAR A) (CAR B) S TYPE
            (LAMBDA (S TYPE)
               (MATCH (CDR A) (CDR B) S TYPE C))))))
((EQ TYPE 'SUBSUMED)
 (COND ((ATOM B) NIL)
       ((VARIABLE B)
        (LET ((VCELL (RASSOC B S)))
             (IF VCELL
                 (IF (EQUAL (CAR VCELL) A)
                     (C S TYPE)
                     NIL)
                 (C (CONS (CONS A B) S) TYPE))))
       ((ATOM A) NIL)
       (T (MATCH (CAR A) (CAR B) S TYPE
            (LAMBDA (S TYPE)
               (MATCH (CDR A) (CDR B) S TYPE C))))))
       (T (BREAK |COMPARE ERROR|))))))
(MATCH A B NIL 'VARIANT (LAMBDA (S TYPE) (LIST TYPE S))))))
```

RASSOC is something of an inverse ASSOC, which searches an

association list for an association whose CDR matches the supplied key.

```
(DEFUN RASSOC (KEY ALIST)
       (DO ((L ALIST (CDR L))) ((NULL L) NIL)
          (COND ((EQUAL KEY (CDAR L)) (RETURN (CAR L))))))
```

UNIFY takes two expressions and a substitution as input. It returns either a list whose first element is a substitution which yields the most general common unifier of the expressions, relative to the given substitution, if they can be unified, or NIL if they cannot be unified.

```
(DEFINE UNIFY
   (LAMBDA (A B ALIST)
      (LABELS ((MATCH
                (LAMBDA (A B S C)
                   (COND ((EQ A B) (C S))
                         ((ATOM A)
                          (COND ((ATOM B)
                                 (IF (AND (NUMBERP A) (NUMBERP B) (= A B)) (C S)))
                                ((VARIABLE B)
                                 (VARSET B A S C))
                                (T NIL)))
                         ((VARIABLE A)
                          (VARSET A B S C))
                         ((ATOM B) NIL)
                         ((VARIABLE B)
                          (VARSET B A S C))
                         (T (MATCH (CAR A) (CAR B) S
                                   (LAMBDA (S)
                                      (MATCH (CDR A) (CDR B) S C)))))))
               (VARSET
                (LAMBDA (VAR NEWVAL S C)
                   (IF (EQUAL VAR NEWVAL) (C S)
                       (LET ((VCELL (ASSOC VAR S)))
                            (IF VCELL (MATCH (CDR VCELL) NEWVAL S C)
                                (FREEFOR VAR NEWVAL S
                                   (LAMBDA ()
                                      (C (CONS (CONS VAR NEWVAL) S)))))))))))
         (MATCH A B ALIST LIST))))
```

```
(DEFINE FREEFOR
   (LAMBDA (VAR EXP SUB CONT)
      (LABELS ((FREELOOP
                (LAMBDA (E C)
                   (COND ((ATOM E) (C))
                         ((VARIABLE E)
                          (IF (EQUAL E VAR) NIL
                              (FREELOOP (CDR (ASSOC E SUB)) C)))
                         (T (FREELOOP (CAR E)
                                      (LAMBDA ()
                                        (FREELOOP (CDR E) C))))))))
         (FREELOOP EXP CONT))))
```

INSTANCE takes an expression and a substitution as input and produces a standardized instance of the expression with that substitution.

```
(DEFINE INSTANCE
   (LAMBDA (EXP SUB)
      (LABELS ((ILOOP
                (LAMBDA (E NEWSUB C)
                   (COND ((ATOM E) (C E NEWSUB))
                         ((VARIABLE E)
                          (LET ((VCELL (ASSOC E NEWSUB)))
                               (IF VCELL (C (CDR VCELL) NEWSUB)
                                   (LET ((VCELL (ASSOC E SUB)))
                                        (IF VCELL
                                            (ILOOP (CDR VCELL) NEWSUB
                                                   (LAMBDA (NEWEXP NEWSUB)
                                                      (C NEWEXP
                                                         (CONS (CONS E NEWEXP)
                                                               NEWSUB))))
                                            (LET ((V (VGENS (CDR E))))
                                                 (C V (CONS (CONS E V)
                                                            NEWSUB)))))))))
                         (T (ILOOP (CAR E) NEWSUB
                                   (LAMBDA (NEWCAR NEWSUB)
                                      (ILOOP (CDR E) NEWSUB
                                             (LAMBDA (NEWCDR NEWSUB)
                                                (C (CONS NEWCAR NEWCDR)
                                                   NEWSUB)))))))))
         (ILOOP EXP NIL (LAMBDA (NEWEXP NEWSUB) NEWEXP)))))
```

THE DISCRIMINATION NETWORK

The following (absurdly hairy) functions implement a discrimination net data base. Ignoring the use of the hash table for the moment, let us first understand how a discrimination network is built. Consider the problem of classifying the S-expression (A (B . C) D). Although internally, this expression is a tree, its structure can be expressed as a string of

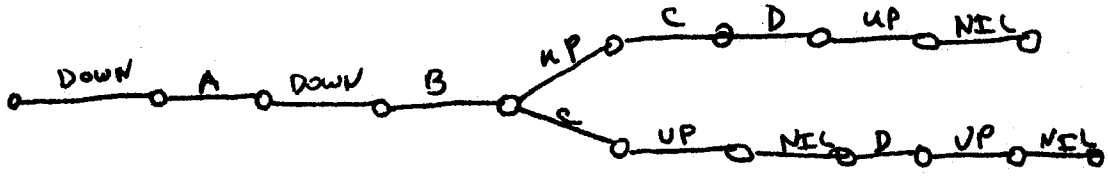
tokens (as for PRINTing it). In this case, the stream of tokens used to discriminate is:

DOWN A *DOWN* B *UP* C D *UP* NIL

A related expression, (A (B C) D), translates into:

DOWN A *DOWN* B C *UP* NIL D *UP* NIL

Given these two expressions, BUCKET would construct a discrimination net with the following structure:



Given any expression, BUCKET extends the discrimination network, if necessary, and returns the bucket represented by the appropriate leaf of the discrimination network.

A variable may appear in any position of an expression to be indexed. Each node of the discrimination network contains a special pointer to the subindex for token streams beginning with a variable.

An interesting complexity in this system is that many structures share the same discrimination subnetworks. We assume the user will use lists to represent logic-like terms. These denote the semantic objects being dealt with. It thus makes sense that EQUAL or VARIANT terms be uniquely represented in the network. This is accomplished by discriminating every non-atomic term from the top of the network and then using the resulting bucket as the token for that term in every stream containing that term. This causes a painful problem: There is now a token for every term, not just every atom. Furthermore, every such token must appear in the top-level node of the network. This makes it unfeasible to use a simple ASSOC of one of these tokens on a part of the node to do a dispatch. Here we introduce a 2-key hash-table to do our associations. Given a token and a discrimination-node, we hash-retrieve an a-list. An element of this a-list beginning with our keys has the required subindex. To introduce further possible bugs, we bubble the association forward in the hash-entry. Donald Duck

```
(PROCLAIM (*EXPR VARIABLE HASH-GET HASH-PUT))
```

The following are special tokens for discriminating through levels of list structure and numbers.

```
(SET' *DOWN* (LIST '*DOWN*))
(SET' *UP* (LIST '*UP*))
(SET' *NUMBER* (LIST '*NUMBER*))
```

DBINIT initializes a supplied variable to contain an empty data base.

```
(DEFINE DBINIT (LAMBDA (DB) (SET DB (LIST NIL NIL))))
```

STUFF retrieves the list of items from a data base bucket.

```
(DEFINE STUFF (LAMBDA (BUCKET) (CAR BUCKET)))
```

INSERT-IN-BUCKET does what it says.

```
(DEFINE INSERT-IN-BUCKET
        (LAMBDA (ITEM BUCKET)
                (RPLACA BUCKET (CONS ITEM (CAR BUCKET)))))
```

BUCKET returns the bucket of items from a data base corresponding to the supplied expression and substitution.

```
(DEFINE BUCKET
   (LAMBDA (EXPRESSION ALIST INDEX)
      (LABELS ((WALK-LIST
                (LAMBDA (FRAGMENT SUBINDEX)
                   (COND ((ATOM FRAGMENT)
                          (GET-SUBINDEX (IF (NUMBERP FRAGMENT)
                                            *NUMBER*
                                            FRAGMENT)
                                        (GET-SUBINDEX *UP* SUBINDEX)))
                         ((VARIABLE FRAGMENT)
                          (LET ((VCELL (ASSOC FRAGMENT ALIST)))
                               (IF VCELL
                                   (WALK-LIST (CDR VCELL) SUBINDEX)
                                   (GET-VARIABLE-SUBINDEX
                                       (GET-SUBINDEX *UP* SUBINDEX)))))
                         (T (WALK-LIST (CDR FRAGMENT)
                                       (WALK-THING (CAR FRAGMENT) SUBINDEX))))))
               (WALK-THING
                (LAMBDA (FRAGMENT SUBINDEX)
                   (COND ((ATOM FRAGMENT)
                          (GET-SUBINDEX (IF (NUMBERP FRAGMENT) *NUMBER* FRAGMENT) SUBINDEX))
                         ((VARIABLE FRAGMENT)
                          (LET ((VCELL (ASSOC FRAGMENT ALIST)))
                               (IF VCELL
                                   (WALK-THING (CDR VCELL) SUBINDEX)
                                   (GET-VARIABLE-SUBINDEX SUBINDEX))))
                         (T (GET-SUBINDEX
                                (WALK-LIST (CDR FRAGMENT)
                                    (WALK-THING (CAR FRAGMENT) INDEX))
                                (GET-SUBINDEX *DOWN* SUBINDEX))))))
               (GET-SUBINDEX
                (LAMBDA (THING INDEX)
                   (LET ((A (HASH-GET INDEX THING)))
                        (IF A (CDR A)
                            (LET ((NEWIND (LIST THING NIL NIL)))
                                 (BLOCK (HASH-PUT NEWIND INDEX)
                                        (RPLACD (CDR INDEX)
                                                (CONS NEWIND (CDDR INDEX)))
                                        (CDR NEWIND))))))))
               (GET-VARIABLE-SUBINDEX
                (LAMBDA (INDEX)
                   (IF (CADR INDEX) (CADR INDEX)
                       (CAR (RPLACA (CDR INDEX) (LIST NIL NIL)))))))
         (WALK-THING EXPRESSION INDEX))))
```

FETCH returns a stream of items from a data base which are candidates for unification with the supplied pattern relative to the supplied substitution. The stream is either NIL, or is a list whose first element is a candidate and whose second element is the continuation (of no arguments) to call to get the next candidate and continuation (or NIL if

none).

FETCH is unpalatably hairy. We apologize. We will award an autographed listing to anyone who has a good explanation of this mess. We also challenge advocates of structured programming disciplines to produce a clearer encoding of the same algorithm. The hair comes from the treatment of variables. Because there may be variables in the fetch patterns and stored expressions, there may be several candidates for a match. We have chosen to generate these incrementally (though the rest of AMORD doesn't require this). This introduces a backtracking system to confuse other matters. More substantively, variables in the fetch pattern must match only well-formed subexpressions. But expressions are recursively defined sequences of tokens; hence the parenthesis grammar must be counted out. Even worse, we allow terminal segments (for example (A . :X)) in both patterns and stored expressions. This leads to a case analysis because the initial conditions of the counting argument have to be considered. UGH!!

```
(DEFINE FETCH
  (LAMBDA (PATTERN ALIST INDEX)
    (LABELS ((WALK-LIST
              (LAMBDA (FRAGMENT SUBINDEX NEXT LOSE)
                (COND ((ATOM FRAGMENT)
                       (GET-SUBINDEX *UP*
                                     SUBINDEX
                                     (LAMBDA (SUBINDEX LOSE)
                                       (GET-SUBINDEX (IF (NUMBERP FRAGMENT)
                                                         *NUMBER*
                                                         FRAGMENT)
                                                     SUBINDEX
                                                     NEXT
                                                     (LAMBDA ()
                                                       (NEXTV NEXT SUBINDEX LOSE))))
                                     LOSE))
                      ((VARIABLE FRAGMENT)
                       (LET ((VCELL (ASSOC FRAGMENT ALIST)))
                            (IF VCELL
                                (WALK-LIST (CDR VCELL) SUBINDEX NEXT LOSE)
                                (GET-VARIABLE-LIST SUBINDEX NEXT LOSE))))
                      (T (WALK-THING (CAR FRAGMENT)
                                     SUBINDEX
                                     (LAMBDA (SUBINDEX LOSE)
                                       (WALK-LIST (CDR FRAGMENT) SUBINDEX NEXT LOSE))
                                     (LAMBDA ()
                                       (GET-SUBINDEX *UP*
                                                     SUBINDEX
                                                     (LAMBDA (SUBINDEX LOSE)
                                                       (NEXTV NEXT SUBINDEX LOSE))
                                                     LOSE)))))))
```

```
(WALK-THING
 (LAMBDA (FRAGMENT SUBINDEX NEXT LOSE)
    (COND ((ATOM FRAGMENT)
           (GET-SUBINDEX (IF (NUMBERP FRAGMENT) *NUMBER* FRAGMENT)
                         SUBINDEX
                         NEXT
                         (LAMBDA () (NEXTV NEXT SUBINDEX LOSE))))
          ((VARIABLE FRAGMENT)
           (LET ((VCELL (ASSOC FRAGMENT ALIST)))
                (IF VCELL
                    (WALK-THING (CDR VCELL) SUBINDEX NEXT LOSE)
                    (GET-VARIABLE-THING SUBINDEX NEXT LOSE))))
          (T (GET-SUBINDEX *DOWN*
                SUBINDEX
                (LAMBDA (SUBINDEX1 LOSE)
                   (WALK-THING (CAR FRAGMENT)
                        INDEX
                        (LAMBDA (SUBINDEX2 LOSE)
                           (WALK-LIST (CDR FRAGMENT)
                                SUBINDEX2
                                (LAMBDA (SUBINDEX3 LOSE)
                                   (GET-SUBINDEX SUBINDEX3
                                                 SUBINDEX1
                                                 NEXT
                                                 LOSE))
                                LOSE))
                        LOSE))
                (LAMBDA () (NEXTV NEXT SUBINDEX LOSE)))))))
(GET-SUBINDEX
 (LAMBDA (THING INDEX NEXT LOSE)
    (LET ((A (HASH-GET INDEX THING)))
         (IF A (NEXT (CDR A) LOSE) (LOSE)))))
```

```
(GET-VARIABLE-LIST
 (LAMBDA (INDEX NEXT LOSE)
   (DUMP INDEX
         (LAMBDA (ASUB LOSE)
           (COND ((EQ (CAR ASUB) *UP*)
                  (DUMP (CDR ASUB)
                        (LAMBDA (ASUB LOSE)
                          (NEXT (CDR ASUB) LOSE))
                        NEXT
                        LOSE))
                 ((EQ (CAR ASUB) *DOWN*)
                  (DUMP (CDR ASUB)
                        (LAMBDA (ASUB LOSE)
                          (GET-VARIABLE-LIST (CDR ASUB)
                                             NEXT
                                             LOSE))
                        (LAMBDA (IND LOSE)
                                (GET-VARIABLE-LIST IND
                                                   NEXT
                                                   LOSE))
                        LOSE))
                 (T (GET-VARIABLE-LIST (CDR ASUB)
                                       NEXT
                                       LOSE))))
         (LAMBDA (VARSIND BARF) (LOSE))
         LOSE)))
(GET-VARIABLE-THING
 (LAMBDA (INDEX NEXT LOSE)
   (DUMP INDEX
         (LAMBDA (ASUB LOSE)
           (COND ((EQ (CAR ASUB) *UP*)
                  (LOSE))
                 ((EQ (CAR ASUB) *DOWN*)
                  (DUMP (CDR ASUB)
                        (LAMBDA (ASUB LOSE)
                          (NEXT (CDR ASUB) LOSE))
                        NEXT
                        LOSE))
                 (T (NEXT (CDR ASUB) LOSE))))
         NEXT
         LOSE)))
```

```
(DUMP
 (LAMBDA (INDEX EA EV LOSE)
    (LABELS ((ALOOP
              (LAMBDA (BL)
                 (COND (BL (EA (CAR BL)
                               (LAMBDA () (ALOOP (CDR BL)))))
                       ((CADR INDEX)
                        (EV (CADR INDEX) LOSE))
                       (T (LOSE))))))
             (ALOOP (CDDR INDEX)))))
(NEXTV
 (LAMBDA (NEXT INDEX LOSE)
         (IF (CADR INDEX)
             (NEXT (CADR INDEX) LOSE)
             (LOSE)))))
(WALK-THING PATTERN INDEX
           (LAMBDA (TERMINAL LOSE)
              (LABELS ((NPOS
                        (LAMBDA (L)
                           (IF L (LIST (CAR L) (LAMBDA () (NPOS (CDR L))))
                               (LOSE)))))
                      (NPOS (CAR TERMINAL))))
           (LAMBDA () NIL)))))
```

The following functions implement the hash table for associations used in making the token dispatch step of the discrimination more efficient.

```
(DECLARE (SPECIAL HASH-ARRAY-SIZE)
         (FIXNUM HASH-ARRAY-SIZE (HASH-NUMBER NOTYPE NOTYPE) NUM)
         (ARRAY* (NOTYPE (HASH-ARRAY ?))))
```

HASH-GET retrieves a specified thing from the hash table of the supplied data base.

```
(DEFUN HASH-GET (INDEX THING)
       (CDR (2-BSSQ INDEX THING
                    (HASH-ARRAY (HASH-NUMBER INDEX THING)))))
```

HASH-PUT inserts a new thing into the hash table of the given data base.

```
(DEFUN HASH-PUT (NEWINDEX INDEX)
       ((LAMBDA (NUM)
                (STORE (HASH-ARRAY NUM)
                       (CONS (CONS INDEX NEWINDEX)
                             (HASH-ARRAY NUM))))
        (HASH-NUMBER INDEX (CAR NEWINDEX))))
```

2-BSSQ searches an association list for an association of the pairing

of the supplied two keys, and for efficiency [Rivest 1976], bubbles the association one step towards the front of the association list.

```
(DEFUN 2-BSSQ (K1 K2 L)
       (PROG (L1 L2)
             (COND ((NULL L) (RETURN NIL))
                   ((AND (EQ K1 (CAAR L)) (EQ K2 (CADAR L)))
                    (RETURN (CAR L))))
             (SET' L2 L)
          LP (SET' L1 (CDR L2))
             (COND ((NULL L1) (RETURN NIL))
                   ((AND (EQ K1 (CAAR L1)) (EQ K2 (CADAR L1)))
                    (RPLACA L2
                            (PROG2 NIL (CAR L1)
                                   (RPLACA L1 (CAR L2))))
                    (RETURN (CAR L2))))
             (SET' L2 (CDR L1))
             (COND ((NULL L2) (RETURN NIL))
                   ((AND (EQ K1 (CAAR L2)) (EQ K2 (CADAR L2)))
                    (RPLACA L1
                            (PROG2 NIL (CAR L2)
                                   (RPLACA L2 (CAR L1))))
                    (RETURN (CAR L1))))
             (GO LP)))
```

This is the ubiquitous number computer.

```
(DEFUN HASH-NUMBER (KEY1 KEY2)
       (\ (BOOLE 6 (MAKNUM KEY1) (MAKNUM KEY2))        ;XOR
          HASH-ARRAY-SIZE))

(SET' HASH-ARRAY-SIZE 1021.)

(ARRAY HASH-ARRAY T HASH-ARRAY-SIZE)
```

This concludes the listing of the interpreter.

Notes

AMORD

A Miracle of Rare Device, a name taken from S. T. Coleridge's poem Kubla Khan. A previous version of AMORD was implemented by Doyle and Steele in the Fall of 1976. That version was based on a threaded, LEAP-like [Feldman and Rovner 1969] data base of triples coupled with an incredibly elaborate system of macros, and was abandoned after the experiences of Steele in writing a rule-based SCHEME complier called CHEAPA, [Steele 1977] and Sussman and Doyle in writing a new version of EL. [Sussman and Stallman 1975] The first version of the interpreter presented here was implemented (without making use of the TMS) by Sussman, de Kleer and Rich for tutorial use in MIT's 6.036 course in the Spring of 1977. This version was then extensively modified by Doyle by integrating the use of the TMS and making various efficiency modifications, and by Sussman in experimenting with successively more refined versions of the discrimination net.

TMS

The Truth Maintenance System is a program developed by Doyle [1977]. Section 3 summarizes its function and use.

SCHEME

SCHEME [Sussman and Steele 1975] is a dialect of LISP with lexical scoping and tail recursion. It proved to be instrumental in writing the discrimination net for AMORD.

MacLISP

MacLISP [Moon 1974] is a powerful dialect of LISP developed by the MIT Artificial Intelligence Laboratory.

Godel

Self-referential facts cannot be recognized, as the order in which rule environments precludes rules with patterns like `(:F (CRETIN :F))`.

Explicit Control

A more detailed discussion of the technique of explicit control encouraged by AMORD can be found in [de Kleer, Doyle, Steele and Sussman 1977].

RABBIT

RABBIT [Steele 1977] is a highly optimizing compiler for SCHEME. RABBIT compiles into a small, machine-language-like subset of MacLISP, which can then be compiled using the MacLISP number compiler to produce very efficient code.

Donald Duck

If you think the structure of our discrimination network is devious, see Drew McDermott's Donald Duck discrimination network!

References

[de Kleer, Doyle, Steele and Sussman 1977]
Johan de Kleer, Jon Doyle, Guy L. Steele Jr., and Gerald Jay Sussman, "Explicit Control of Reasoning," MIT AI Lab, Memo 427, June 1977.

[Doyle 1977]
Jon Doyle, "Truth Maintenance Systems for Problem Solving," MIT AI Lab TR-419, June 1977.

[Feldman and Rovner 1969]
Jerome A. Feldman and Paul D Rovner, "An Algol-Based Asssociative Language," *CACM 12*, #8, (August 1969), pp. 439-449.

[Moon 1974]
David A. Moon, "MacLISP Reference Manual," MIT Project Mac, Revision 0, April 1974.

[Rivest 1976]
Ronald Rivest, "On Self-Organizing Sequential Search Heuristics," *CACM 19*, #2, (February 1976), pp. 63-67.

[Steele 1977]
Guy L. Steele Jr., "Compiler Optimization Based on Viewing LAMBDA as RENAME plus GOTO," MIT SM Thesis, Electrical Engineering and Computer Science, May 1977.

[Sussman and Stallman 1975]
Gerald Jay Sussman and Richard Matthew Stallman, "Heuristic Techniques in Computer-Aided Circuit Analysis," *IEEE Transactions on Circuits and Systems*, Vol. CAS-22, No. 11, November 1975, pp. 857-865.

[Sussman and Steele 1975]
Gerald Jay Sussman and Guy Lewis Steele Jr., "SCHEME: An Interpreter for Extended Lambda Calculus," MIT AI Lab Memo 349, December 1975.