# TMYCIN* EXPERT SYSTEM TOOL

GORDON S. NOVAK, JR.

APRIL 1987 AI87-52
(REVISED AUGUST 1988)

---

* TMYCIN was originally written for use in a course developed by faculty of the University of Texas for Hewlett Packard.

† The Artificial Intelligence Laboratory at the University of Texas receives major support from the U.S. Army Research Office under contract DAAG29-84-K-0060. The A.I. Lab has also benefitted from major equipment grants by Hewlett Packard and Xerox.

# Table of Contents

# TMYCIN Expert System Tool

Gordon S. Novak Jr.
Computer Science Department
University of Texas at Austin
Austin, TX 78712

## 1. Introduction

TMYCIN ("Tiny EMYCIN") is a simple expert system tool patterned after the EMYCIN tool developed at Stanford [Shortliffe 76] [EMYCIN 81]. TMYCIN does not attempt to provide all of the features of EMYCIN; it is intended to provide some of the most commonly used features in a package that is small and simple. The internal implementation of TMYCIN has been written from scratch and is therefore different from that of EMYCIN.

## 2. Data

Data about a particular case is stored in a record structure called a *context*; the current context is referenced in rules by the global variable `cntxt`. TMYCIN allows only a single level of context.

### 2.1. Context Description

The data in a context are described in a *context description* or *class*, which is defined to the system by a call to DEFCONTEXT:

```
(defcontext <context-name>
            <parameters>
            <initial-data>
            <goals>)
```

where `<context-name>` is the name of the context (usually the name of the kind of object being identified or diagnosed, e.g., ROCK or PATIENT), `<parameters>` is a list of parameter descriptions, `<initial-data>` is a list of the parameters whose values are to be asked for at the start of every consultation, and `<goals>` is a list of parameters

whose values are sought as the result of the consultation. An example of a call to DEFCONTEXT is shown below.

```
(defcontext 'rock                                  ; Context Name
            '((color (brown black white))          ; Parameters
              (hardness posnumb)
              (environment (igneous metamorphic sedimentary)
                 ("What is the type of geologic environment"
                  "in which the specimen was found?"))
              (identity atom)
              (pretty nil))                        ; Yes/No parm
            '(color)                               ; Initial data
            '(identity))                           ; Goals
```

Each <parameter> description is a list of three items:

```
(<parameter-name> <type> <prompt>)
```

1. The <parameter-name> is always a single symbol, e.g. COLOR.

2. The <type> of the parameter may be:

   a. A type as used in EMYCIN, e.g. POSNUMB (positive number). This is not checked, but is shown to the user when prompting for user input.

   b. A list of possible values, e.g. (BROWN BLACK WHITE). This is not checked, but is shown to the user when prompting for user input.

   c. NIL, which indicates a Yes/No parameter. This value is examined by the input routine, which will convert an input of NO to (YES -1.0) if the type is NIL. A type of NIL will also affect the way in which the parameter is described when prompting for input.

3. The <prompt> (optional) may be either a string or a list of strings. The prompt string(s) are shown to the user when prompting for input, or in response to a "?" input; the purpose is to explain in more detail the input that is being asked for (including, for example, units of measurement).

### 2.2. Internal Data Storage

A context in TMYCIN is implemented as a Lisp symbol (a GENSYM symbol formed from the context name, e.g. ROCK37) with the data values stored on its property list. A pointer to the context description is stored under the property name ISA.

Each parameter value is stored as a list of values and certainty factors, e.g.

```
COLOR ((RED 0.4)(WHITE 0.3)(BLUE 0.1))
```

where COLOR is the parameter/property name and the values are stored as a list under that property. The values are sorted by certainty factor (CF), with the value having the highest CF first.

## 3. Rules

### 3.1. Rule Format

Rules are defined to the system using the function DEFRULES, which takes one or more unquoted rules as input. Each rule has the format:

```
(<rulename> <premises>
            <conclusion>)
```

Usually the <premises> is a conjunction of conditions grouped within a call to the function $AND , and the conclusion is a call to the function CONCLUDE (the function DO-ALL can be used for multiple conclusions or for actions in addition to calling CONCLUDE). An example of a rule definition is shown below.

```
(defrules

(rule101 ($and (same cntxt color black)
               (notsame cntxt pretty yes)
               ($or (between* (val1 cntxt hardness) 3 5)
                    (same cntxt environment sedimentary)))
         (conclude cntxt identity coal tally 400))

)
```

### 3.2. $AND and $OR

The condition part of a rule is usually formed from tests of parameter values, combined by the functions **$AND** and **$OR**. **$AND** evaluates each of its clauses in order. If any clause returns **NIL** or returns a CF (*certainty factor*) value less than the .2 threshold, **$AND** immediately returns **NIL** (without evaluating any other clauses); otherwise, **$AND** returns the minimum CF that any clause returned. **$OR** is "true" if any of its clauses is "true"; **$OR** returns the maximum CF returned by any of its clauses. **$AND** and **$OR** may be nested.

Before trying all of its clauses, **$AND** does a "prescan" of the clauses to see if any of them is already known to be false (CF of **NIL** or below the .2 threshold); if so, **$AND** returns **NIL** without evaluating any of the other clauses.

### 3.3. Testing Data Values

Several functions are provided to test data values within rules. These differ in terms of the certainty factor required to make the condition "true" and in terms of the CF value returned. Each of these functions uses the global variable **CNTXT**, which is a pointer to the current context.

- **(SAME CNTXT <parameter> <value>)** tests whether the specified parameter has the specified value with CF $>$ .2 ; the value returned is the CF of the parameter. A Yes/No parameter is tested for the value "Yes" with **(SAME CNTXT <parameter> YES)**.

- **(NOTSAME CNTXT <parameter> <value>)** tests whether the specified parameter does not have the specified value at all, or has it with CF $<=$ .2 ; if so, the value returned is 1.0 . Note that **NOTSAME** returns "true" for a parameter whose value is "unknown".

- **(THOUGHTNOT CNTXT <parameter> <value>)** tests whether the specified parameter has the specified value with CF $<$ -.2 ; if so, the value returned is the negative of the CF value. **THOUGHTNOT** is the negative counterpart of **SAME**. A Yes/No parameter is tested for the value "No" with **(THOUGHTNOT CNTXT <parameter> YES)**. **THOUGHTNOT** requires a negative value for the specific parameter; it will not respond to an "unknown" value.

- **(KNOWN CNTXT <parameter>)** tests whether the specified parameter has a

value with CF > .2 (or, for yes/no parameters, CF < -.2); if so, the value returned is always 1.0, which means that the CF of data tested using KNOWN will not affect the CF returned by $AND.

- (NOTKNOWN CNTXT <parameter>) tests whether the specified parameter has no value with CF > .2 (or, for yes/no parameters, CF < -.2); if so, the value returned is 1.0 . This predicate may be used to test for data that the user specifies as "unknown".

### 3.4. Numeric Tests

In order to perform numeric tests or other calculations with parameter values, it is first necessary to get the numeric values of the parameters; this is done using the function VAL1. VAL1 gets the value of a parameter which has the highest CF of all the possible values stored for that parameter. The format is:

```
(VAL1 CNTXT <parameter>)
```

If the parameter has a numeric value, VAL1 will return that value, which can then be used in numeric calculations or in the numeric comparison functions described below.

The numeric comparison functions provided with TMYCIN are different from the plain Lisp comparison functions in two respects: they are able to tolerate non-numeric arguments (in which case they return NIL), and they return a value of 1.0 if the test is "true" so that they can be used within $AND. The comparison functions are:

```
(greaterp* <numexp1> <numexp2>)
(greateq*  <numexp1> <numexp2>)
(lessp*    <numexp1> <numexp2>)
(lesseq*   <numexp1> <numexp2>)
(between*  <numexp1> <numexp2> <numexp3>)
```

BETWEEN* tests whether <numexp2> <= <numexp1> < <numexp3>.

## 4. Input

When TMYCIN asks for a parameter value during a consultation, there are several kinds of response the user can give:

1. The user can simply enter a single data value, e.g. BLUE. The resulting stored value will be a list of that one value with a certainty of 1.0, i.e.,

((BLUE 1.0)). For a Yes/No parameter, the user may enter YES, Y, NO, or N; NO and N are converted to ((YES -1.0)) .

2. The user can enter a list of a single value and a certainty factor, e.g., (YES 0.6). The resulting stored value will be a list of that one value, i.e., ((YES 0.6)).

3. The user can enter a list of multiple values and certainty factors, e.g., ((RED 0.5)(ORANGE 0.5)). TMYCIN does not enforce any kind of consistency among certainty factors. Therefore, a parameter that is thought of as multivalued may have several parameters with high certainty, e.g., ((RED 1.0) (WHITE 1.0) (BLUE 1.0))

4. The user can enter UNK or UNKNOWN if the value of the parameter is unknown. This results in a "dummy" data set, ((UNKNOWN 0.0)) . In general, unknown values will not be considered as "true" by the predicates that test data values (with the exception of NOTSAME and NOTKNOWN), so that most rules involving unknown data will not fire.

5. The user can enter ?. The system will respond by printing the prompt string for the parameter (if there is one) and the type specified for the parameter. Then the user will be asked for the parameter value again. Note that with *printdes* set to T, the default value, this information is printed automatically; the ? input is mainly useful when *printdes* is set to NIL.

6. The user can enter WHY. The system will respond by printing the rule that is currently being examined, then ask for the parameter value again.

### 4.1. Input Options

The global variable *printdes* determines whether the data type and prompt information will be shown to the user automatically when asking for input. The default value of *printdes* is T.

If desired, the user may define a function to obtain input, either from the user or from another source (e.g., a database or special I/O device). If the parameter name has the property ASKFN defined, the function specified for the ASKFN property will be called to obtain the value; its parameters are the data context and the parameter name. The ASKFN should return a list of (<value> <cf>) pairs, as described above. For

example, if an ASKFN reads a VOLTAGE of 4.6 volts from an A/D converter, it should return ((4.6 1.0)) .

```
; Specify the 'ask' function for voltage
(setf (get 'voltage 'askfn) #'readvoltage)

; Simulate reading of voltage: 4 volts + 0-1 volts noise
(defun readvoltage (cntxt parm)
  (list (list (+ 4.0 (random 1.0)) 1.0)) )
```

In some cases, it may be desirable to ask the user for a value before trying to infer a parameter value using rules. For example, the TEMPERATURE of a patient will usually be known. If the ASKFIRST property is defined with a non-NIL value for a parameter name, then the user will be asked for a value before rules to infer the parameter are tried.

```
(setf (get 'temperature 'askfirst) t)
```

## 5. Escaping to Lisp

An important feature of an expert system tool is the ability to escape from it into the underlying programming language to perform computations that are not easily supported by the tool. TMYCIN provides several places at which an expert system application can escape into Lisp.

### 5.1. Special Input Functions

As mentioned above, it is possible to put a function name on the property list of a parameter name as the value of the ASKFN property. This will cause that function to be called for input of that parameter rather than asking the user for the value. This allows some or all of the input data used by the expert system to be gotten from other sources, such as databases, data files, or special input devices. If desired, the parameters could also be specified as ASKFIRST parameters so that data will be obtained for them before the consultation begins.

### 5.2. Functions in Rule Premises

A call to an arbitrary function can be included in the premise of a rule. Such a function may use the global variable CNTXT to refer to the current data context. The function should return a value which is a certainty factor (a number from -1.0 to 1.0), or NIL to indicate failure. Consult the existing predicate functions for examples. If no value is known for a parameter and the global variable *prescan* is non-NIL, the value 1.0 should be returned.

### 5.3. Calculations in Rule Conclusions

If the <value> part of a CONCLUDE call is a single atom (symbol), it is treated as if it were quoted:

```
(conclude cntxt identity coal tally 400)
```

In this case, coal is treated as a quoted value. However, if the <value> is a list, it is evaluated; this allows calculations to be performed in the conclusion part of a rule. The function VAL1 can be used to get parameter values.

```
(rule107 ($and (same cntxt shape circle)
               (known cntxt radius))
         (conclude cntxt area
                   (* 3.14159
                      (expt (val1 cntxt radius) 2))
                   tally 1000))
```

### 5.4. Calculations in Rule Certainty Factors

In some cases, it is desirable to use data values in calculating certainty factors. For example, suppose that a physician expert specifies that lung cancer is ruled out for patients less than 20 years old, and is to be linearly weighted negatively for patients from 20 to 30 years old.

```
(rule112 (lesseq* (val1 cntxt age) 30)
         (conclude cntxt diagnosis lung-cancer
                   tally (if (< (val1 cntxt age) 20)
                             -1000
                             (* (- 30 (val1 cntxt age))
                                -100))) )
```

### 5.5. Calling Functions From Conclusion

It is sometimes useful to call user functions in the conclusion part of a rule. The function DO-ALL is used to perform multiple actions in the conclusion part of a rule. By putting both a CONCLUDE call and a call to a user function within the DO-ALL, the user function can be called when the rule fires. At least one CONCLUDE call is needed in order to cause the rule to be considered during backchaining; the parameter that is CONCLUDEd should either be a goal parameter or one that will be traced in seeking the value of a goal parameter.

```
(rule109 ($and (same cntxt smoke yes)
               (same cntxt heat yes))
         (do-all (conclude cntxt problem fire tally 800)
                 (sound-fire-alarm-fn)))
```

## 6. Notes on Using TMYCIN

### 6.1. Deleting Rules

In most cases, when a rule is edited and defined again using DEFRULES, no further action is necessary. However, if the rule conclusion is changed so that it no longer concludes the same parameter, or if the rule is to be deleted entirely, special action is necessary. The reason for this is that rules are indexed so that each parameter has a list of the rules that conclude it; if a rule is to be deleted, it must be removed from this list.

`(delrule <rulename>)`[1] will delete the single rule `<rulename>`.

`(clear-rules)` will delete all rules.

---

[1]These functions were written by Hiow-Tong Jason See.

### 6.2. Self-Referencing Rules

Sometimes rules are written that reference the same parameter in both the premise and conclusion; these might be used, for example, to increase the certainty of a parameter that is already concluded if additional factors are present. In order to run such rules correctly, all other rules that conclude the parameter must run first. To make this happen, all self-referencing rules should be put at the *end* of the file of rules; this will make them run last in the ordering used by TMYCIN.

```
(rule119 ($and (same cntxt identity rattlesnake)
               (same cntxt bit-someone yes)
               (same cntxt victim-died yes))
         (conclude cntxt identity rattlesnake tally 800))
```

## 7. Useful Functions

### 7.1. Starting the Consultation

DOCONSULT is used to start a new consultation; its optional parameter is a context class name.

```
(doconsult)

(doconsult 'rock)
```

### 7.2. Why Questions

Two functions, WHY and WHYNOT, are provided to allow the user to ask about the system's reasoning at the end of a consultation. WHY will print out the rule(s) that concluded values for a parameter. WHYNOT asks why a particular conclusion was *not* reached. It prints rule(s) that might have reached the specified conclusion along with the failing condition that prevented each rule from firing.

Both functions allow omission of parameters for convenience. The full set of parameters is:

```
(why     <context> <parameter> <value>)

(whynot <context> <parameter> <value>)
```

It is permissible to omit the first parameter, the first two, or all three (in the case of the WHY function). The <context> defaults to the context for the most recently run consultation. The <parameter> defaults to the first parameter in the GOALS list. The <value> defaults to the most strongly concluded value for the <parameter>. Example calls are:

```
(why)
(why obsidian)
(why identity obsidian)
(why rock37 identity obsidian)

(whynot coal)
(whynot identity coal)
(whynot rock37 identity coal)
```

### 7.3. Translation of Rules into English

The function ENGLRULE translates a previously defined rule into an English-like format. Its argument is a quoted rule name:

```
(englrule 'rule101)
```

produces:

```
If:
      1) the COLOR of the ROCK is BLACK, and
      2) PRETTY is not true of the ROCK, and
      3)    1) the HARDNESS of the ROCK is 4, or
            2) the ENVIRONMENT of the ROCK is SEDIMENTARY
then:
      there is weakly suggestive evidence (0.4)
      that the IDENTITY of the ROCK is COAL
```

### 7.4. Analysis of the Knowledge Base

SHOWRULE will pretty-print a rule in internal (Lisp) form.

```
(showrule 'rule101)
```

LISTPARMS[2] will print out the parameters defined for the current context class (gotten from the current value of CNTXTNAME , which will be set to the context name used in the last call to DEFCONTEXT).

LISTRULES will list all the rules, in both English and internal forms.

ANALYZE-KB will analyze the knowledge base, listing the rules that can conclude each parameter.

LISTKB will print a complete listing of the knowledge base, i.e., it does ANALYZE-KB, LISTPARMS, and LISTRULES.

### 7.5. Other Useful Functions

SHOWPROPS will pretty-print the property list of an atom, which is useful for looking at the data values stored for a particular consultation:

```
(showprops 'rock37)
```

## 8. How TMYCIN Works

The basic operation of TMYCIN is very simple. The function **doconsult**, which is the top-level function that performs a consultation, first makes a new symbol from the context name (e.g., ROCK37) to be the context for the consultation. Next, it asks the user for values of the parameters that are specified as initial-data parameters in the context definition. Then it calls **bc-goal** to find values for each of the goal parameters. Finally, it prints the results.

---

[2]These functions were written by Hiow-Tong Jason See.

The basic function to find values for a parameter (called *tracing* the parameter) is `bc-goal`, so named because it *backchains* on rules to try to find values for its goal parameter. `bc-goal` finds the set of rules that could conclude some value for the desired parameter and runs each of them; if there are no rules to conclude the value of a parameter, it asks the user for the value. The rules that can conclude each parameter are stored on the property list of the parameter under the property `RULES`; this cross-indexing is done by `DEFRULES` when the rules are defined. For example, `RULE101`, which concludes that the identity of a rock is coal, is stored as one of the rules under the property `RULES` of the symbol `IDENTITY`. Note that while `bc-goal` will often be called by a predicate seeking a particular value for a parameter, e.g. `(SAME CNTXT COLOR WHITE)`, it will try to run all rules that conclude *any* value for the parameter (in this case, `COLOR`) before it quits.

Running a rule is done in two stages. First, the antecedent (left-hand side or "if" part) of the rule is evaluated; if it has a "true" value, i.e., a CF greater than 0.2, the consequent ("then" part) of the rule is executed. Evaluating the antecedent will typically involve tests of parameter values using predicates such as `SAME`; these predicates call the function `parmget` to get the value(s) of the parameter. `parmget` returns an existing value if there is one; otherwise, it calls `bc-goal` to find the value. In this way, `bc-goal` does a depth-first search of the rule tree, where the root(s) of the tree are goal parameters and the terminal nodes of the tree are parameter values provided by the user as input.

## 9. References

[EMYCIN 81] Van Melle, W., Scott, A. C., Bennett, J. S., Peairs, M.
*The Emycin Manual.*
Technical Report STAN-CS-81-885, Computer Science Dept., Stanford University, Oct, 1981.

[Shortliffe 76] Shortliffe, E.H.
*Computer Based Medical Consultations: MYCIN.*
American Elsevier, 1976.

```
; tmycin.lsp        28 Oct 1988          Common Lisp version
;
; TMYCIN = Tiny EMYCIN-like Expert System Tool
;
; Gordon Novak, CS Dept, University of Texas at Austin, 78712
;
; Copyright (c) 1988 by Gordon S. Novak Jr.
; This program may be freely copied, used, or modified,
; provided that this copyright notice is included in each
; copy of the code or parts thereof.
;
; The Artificial Intelligence Laboratory of the University of
; Texas at Austin receives major support from the U.S. Army
; Research Office under contract DAAG29-84-K-0060.
; The AI Lab has also benefitted from major equipment grants by
; Hewlett Packard and Xerox.
;
; This program, originally developed for use in a course for
; Hewlett Packard, simulates the backchaining expert system tool
; EMYCIN.  Obviously, given the size of this system, many
; features of EMYCIN are not supported; however, it is suitable
; for student exercises.  The program is written from scratch;
; the internals are different from those of EMYCIN.
;
; Input of parameter values may be in several forms:
;      RED                        one value with CF = 1.0
;      (RED 0.6)                  one value with specified CF
;      ((RED 0.3)(BLUE 0.3))      multiple values and CF's
;      ?                          Ask for explanation.
;
; Internal Representation:
;
; Context: a Gensym atom with an ISA pointer to the Class atom,
;          PARENT pointer to higher-level context.
;          (Note: only a single level of context is implemented.)
;
; Data values are stored on property list under the parameter
; name as as a list of (Value CF) pairs.
;
; Additions:  Improved Printconclusion (Jeff Nowell, HP).
;             Fix macro-expansion problem in Gold Hill
;               (Bob Causey).
;             Auxiliary functions (Hiow-Tong Jason See).
;(in-package :user)

(defvar allrules nil)          ;  List of all rules, for user.
(defvar cntxt nil)             ;  Current data context
(defvar topctx nil)            ;  Top context of consultation
(defvar cntxtname nil)         ;  Name of the top context class
(defvar tally 0)               ;  For CF calculations
(defvar prevgoals nil)         ;  Loop prevention in bc-goal
(defvar runrulerulename nil)   ;  Used in answering why question
(defvar *printdes* t)          ;  To always print parm prompt
(defvar *prescan* nil)         ;  To preview rule antecedent
```

```
(setq cntxt nil)
(setq tally nil)
(setq allrules nil)

; Copy and then eval.  This crock is included for Gold Hill Lisp,
; which bashes macro calls.  For better Lisps, eval can be used.
(defun tmycin-eval (x) (eval (copy-tree x)))

;----------  Functions for defining the Knowledge Base ----------

; Define a context class.
; Args are: context name, parameters, initialdata, goals.  Each
; parameter spec is a list of parameter name, expected type, and
; prompt string.  NIL is the expected type for YES/NO parameters.
(defun defcontext (contextname parms initialdata goals)
    (setq cntxtname contextname)
    (setf (get contextname 'parameters) parms)
    (setf (get contextname 'initialdata) initialdata)
    (setf (get contextname 'goals) goals)
    contextname)

; Define a set of rules
(defmacro defrules (&rest lvarfordefrules)
  '(progn (mapc #'defrule (quote ,lvarfordefrules)) t))

; Define a new rule, index on conclusions, add to ALLRULES.
; A rule looks like: (RULE101 ($AND ...) (CONCLUDE ...))
(defun defrule (rule)
  (let (parms concpart rulename)
    (setq rulename (if (eq (car rule) 'if) (make-atom 'rule)
                       (car rule)))
    (setf (get rulename 'rule) rule)
    (setq concpart (caddr rule))
    (setq parms (cond ((eq (car concpart) 'conclude)
                       (list (caddr concpart)))
                      ((eq (car concpart) 'do-all)
                       (mapcan #'(lambda (conc)
                                   (if (eq (car conc) 'conclude)
                                       (list (caddr conc)) ))
                               (cdr concpart)) )
                      (t (error "Bad conclusion part of rule ~A"
                                rule))) )
    (mapc #'(lambda (parm)
              (unless (member rulename (get parm 'rules))
                (setf (get parm 'rules)
                      (nconc (get parm 'rules)
                             (list rulename)))))
          parms)
    (unless (member rulename allrules)
      (setq allrules (nconc allrules (list rulename)))) ))

;----------  Functions to run a Consultation  ------------------

; Run a consultation.  E.g., (DOCONSULT 'ROCK)
```

```
(defun doconsult (&optional topclass)
  (let (goals)
    (unless topclass (setq topclass cntxtname))
    (terpri) (terpri)
    (setq goals (get topclass 'goals))
    (setq topctx (make-context topclass nil))
    (getinitialdata topctx)
    (setq *prescan* nil)
    (mapc #'(lambda (goal) (bc-goal topctx goal (list goal)))
          goals)
    (format t "~%~%The conclusions for ~A are as follows:~%"
              topctx)
    (mapc #'(lambda (goal) (printconclusion topctx goal)) goals)
    topctx ))

; Make a new data context of CLASS with a pointer to PARENT
; context.  Example:  CLASS = CULTURE and PARENT = PATIENT1
(defun make-context (class parent)
  (let ((cntxt (make-atom class)))
    (setf (get cntxt 'isa) class)
    (if parent (setf (get cntxt 'parent) parent))
    cntxt ))

; Make a gensym atomname with a specified name.
(defun make-atom (name)
  (let (newatom)
    (setq newatom
          (intern (symbol-name (gensym (symbol-name name)))))
; If this atom has something on its proplist, try again.
    (if (or (get newatom 'rule) (get newatom 'isa))
        (make-atom name)
        newatom) ))

; Get initial data for a class
(defun getinitialdata (cntxt)
  (let ((class (get cntxt 'isa)))
    (mapc #'(lambda (parm)
              (storeparmvals cntxt parm (askvalue cntxt parm)))
          (get class 'initialdata)) ))

; Backchain through rules to try to conclude goal
(defun bc-goal (cntxt parm prevgoals)
  (prog ((cntxtp (findparmctx cntxt parm)) rules asked)
; If parm has already been traced, don't do it again.
    (cond ((get cntxtp parm))
          (*prescan* (return nil)))
    (setq rules (get parm 'rules))
    (if (setq asked (or (null rules) (get parm 'askfirst)))
        (storeparmvals cntxtp parm (askvalue cntxtp parm)))
    (unless (hasvalue cntxtp parm)
            (mapc #'(lambda (rule) (runrule cntxt rule)) rules))
    (cond ((hasvalue cntxtp parm))
          ((not asked)
          (storeparmvals cntxtp parm (askvalue cntxtp parm)))
```

```
          (t (setf (get cntxtp parm)
                   (list (list 'unknown 0.0)))))
    (return (get cntxtp parm)) ))

; Find proper level of context for a parameter
(defun findparmctx (ctxt parm)
  (let ((class (get ctxt 'isa)))
    (if (and ctxt class)
        (if (assoc parm (get class 'parameters))
           ctxt
           (findparmctx (get ctxt 'parent) parm))
       (if ctxt (error "Context has no Class")
          (error "Parm ~A misspelled or left out of defcontext"
                 parm))) ))

; Ask for a parameter value
(defun askvalue (cntxt parm)
  (let ((askfn (get parm 'askfn)))
    (if askfn (funcall askfn cntxt parm) (askuser cntxt parm)) ))

; Ask the user for a data value
(defun askuser (cntxt parm)
  (prog (inp parmdes)
    (setq parmdes (findparmdes cntxt parm))
top (askparmquestion cntxt parm parmdes)
    (if *printdes* (progn (terpri) (printparmdes parmdes)))
lp  (terpri)
    (setq inp (read))
    (cond ((eq inp '?) (printparmdes parmdes) (go lp))
          ((eq inp 'why) (terpri)
           (princ "We are examining the following rule:")
           (terpri) (terpri) (showrule runrulerulename)
           (terpri) (go top))
          ((member inp '(y yes n no))
           (if (null (cadr parmdes))       ; test for yes/no parm
               (setq inp (list 'yes (if (member inp '(n no))
                                        -1.0 1.0)))))
          ((member inp '(unk unknown))
           (setq inp (list 'unknown 0.0))) )
    (return (if (consp inp)
                (if (consp (car inp)) inp (list inp))
                (list (list inp 1.0)) )) ))

; Find description for a parameter
(defun findparmdes (ctxt parm)
  (let (class)
    (if (and ctxt (setq class (get ctxt 'isa)))
        (or (assoc parm (get class 'parameters))
           (findparmdes (get ctxt 'parent) parm))
       (if ctxt (error "Context has no Class")
          (error "Parm ~A misspelled or left out of defcontext"
                 parm))) ))

; Print description of a parameter
```

```
(defun printparmdes (parmdes)
  (if (cddr parmdes)
    (mapc #'(lambda (x) (spaces 4) (princ x) (terpri))
          (if (consp (caddr parmdes)) (caddr parmdes)
              (list (caddr parmdes)))))
  (format t "    Expected values are: ~A~%"
    (or (cadr parmdes) "Yes/No")) )

; Ask the question associated with parameter parm.
(defun askparmquestion (cntxt parm parmdes)
  (format t (if (cadr parmdes) "What is the ~A of ~A?"
                               "Is ~A true of ~A?")
    parm cntxt) )

; Store values for a parameter, sorting by CF.
(defun storeparmvals (ctxt parm vals)
  (setf (get ctxt parm)
        (sort vals #'(lambda (u v) (> (cadr u) (cadr v))) ) ) )

; Test if a value is defined for a parameter
(defun hasvalue (ctxt parm)
  (let ((vals (get ctxt parm)))
    (and vals (not (eq (caar vals) 'unknown))) ))

; Run a rule: if antecedent is true, execute consequent.
(defun runrule (cntxt rulename)
  (let ((rule (get rulename 'rule))(runrulerulename rulename)
        tally)
    (setq tally (tmycin-eval (cadr rule)))
    (if (and tally (numberp tally) (> tally 0.2))
       (tmycin-eval (caddr rule))) ))

;----------  Functions for Left-Hand Side of a Rule  ------------

; AND together clauses so long as CF > .2; return minimum CF.
(defmacro $and (&rest conditions)
  '($andexpr (quote ,conditions)))
(defun $andexpr (clauses)
  (let (pre)
    (setq *prescan* t)
    (setq pre ($andexprb clauses 1.0))
    (setq *prescan* nil)
    (if pre ($andexprb clauses 1.0)) ))
(defun $andexprb (clauses cf)
  (let (cr)
    (if clauses (if (setq cr (eval (car clauses)))
                    (if (and (numberp cr) (> cr 0.2))
                        ($andexprb (cdr clauses) (min cf cr))))
                cf) ))

; OR together clauses, return maximum CF, stop on CF = 1.0.
(defmacro $or (&rest conditions)
  '($orexpr (quote ,conditions) nil))
(defun $orexpr (clauses cf)
```

```
  (let (cr)
    (if clauses
        (if (setq cr (eval (car clauses)))
           (setq cf (if cf (if (numberp cr) (max cf cr) cf)
                       cr))))
    (if (and (numberp cf) (= cf 1.0)) cf
        (if (cdr clauses) ($orexpr (cdr clauses) cf) cf)) ))

; Get a parameter value, backchaining for it if needed.
(defun parmget (cntxt parm)
  (or (cntxtget cntxt parm)
      (unless (member parm prevgoals)          ; to prevent loops
            (bc-goal cntxt parm (cons parm prevgoals)))) )

; Get value of parameter from a context, looking up parent chain
(defun cntxtget (cntxt parm)
   (if cntxt (or (get cntxt parm)
                 (cntxtget (get cntxt 'parent) parm))) )

; Test for a specified value with CF > .2, return CF.
(defmacro same (cntxt parm value)
  '(sameexpr ,cntxt (quote ,parm) (quote ,value)))
(defun sameexpr (cntxt parm value)
  (let ((vals (parmget cntxt parm)) pair)
    (if (and (setq pair (assoc value vals)) (> (cadr pair) 0.2))
        (cadr pair)
       (if (and (null vals) *prescan*) 1.0)) ))

; Test for a specified value with CF <= .2, return 1.0
(defmacro notsame (cntxt parm value)
  '(notsameexpr ,cntxt (quote ,parm) (quote ,value)))
(defun notsameexpr (cntxt parm value)
  (let ((vals (parmget cntxt parm)) pair)
    (if (or (null (setq pair (assoc value vals)))
            (<= (cadr pair) 0.2))
        1.0) ))

; Test for a specified value with CF < -.2, return -CF.
(defmacro thoughtnot (cntxt parm value)
  '(thoughtnotexpr ,cntxt (quote ,parm) (quote ,value)))
(defun thoughtnotexpr (cntxt parm value)
  (let ((vals (parmget cntxt parm)) pair)
    (if (and (setq pair (assoc value vals)) (< (cadr pair) -0.2))
        (- (cadr pair))
       (if (and (null vals) *prescan*) 1.0)) ))

; Test for some value with CF > .2, return 1.0
(defmacro known (cntxt parm)
  '(knownexpr ,cntxt (quote ,parm)))
(defun knownexpr (cntxt parm)
  (let ((vals (parmget cntxt parm))
        (des (findparmdes cntxt parm)))
    (if (some #'(lambda (pair) (or (> (cadr pair) 0.2)
                                   (and (null (cadr des))
```

```
                                        (< (cadr pair) -0.2))))
             vals)
        1.0
       (if (and (null vals) *prescan*) 1.0)) ))

; Test for no value with CF > .2, return 1.0 if there is none.
(defmacro notknown (cntxt parm)
  '(notknownexpr ,cntxt (quote ,parm)))
(defun notknownexpr (cntxt parm)
  (let ((vals (parmget cntxt parm))
        (des (findparmdes cntxt parm)))
    (unless (some #'(lambda (pair)
                      (or (> (cadr pair) 0.2)
                          (and (null (cadr des))
                               (< (cadr pair) -0.2))))
                  vals)
            1.0) ))

; Get the value of a parameter.
; The value with highest CF is returned.
(defmacro val1 (cntxt parm)
  '(val1expr ,cntxt (quote ,parm)))
(defun val1expr (cntxt parm) (caar (parmget cntxt parm)) )

; EMYCIN-style comparisons
(defun greaterp* (x y)
  (if (and x y (numberp x) (numberp y) (> x y)) 1.0
      (if (and *prescan* (or (null x) (null y))) 1.0) ))

(defun greateq* (x y)
  (if (and x y (numberp x) (numberp y) (>= x y)) 1.0
      (if (and *prescan* (or (null x) (null y))) 1.0) ))

(defun lessp* (x y)
  (if (and x y (numberp x) (numberp y) (< x y)) 1.0
      (if (and *prescan* (or (null x) (null y))) 1.0) ))

(defun lesseq* (x y)
  (if (and x y (numberp x) (numberp y) (<= x y)) 1.0
      (if (and *prescan* (or (null x) (null y))) 1.0) ))

(defun between* (x low high)
  (if (and x low high (numberp x) (numberp low) (numberp high)
           (<= low x) (< x high))
      1.0
      (if (and *prescan* (or (null x) (null low) (null high)))
          1.0) ))

;----------  Functions for Right-Hand Side of a Rule  -----------

; Basic rule conclusion function
;  (CONCLUDE CNTXT parm val TALLY cf)
(defmacro conclude (cntxt parm value tally rulecf)
  '(concludeexpr ,cntxt (quote ,parm) (quote ,value)
```

```
                ,tally ,rulecf))
(defun concludeexpr (cntxt parm val tally rulecf)
  (let ((cntxtp (findparmctx cntxt parm)) vals
        pair (newcf (* tally (/ (float rulecf) 1000.0)))
        (vall (if (atom val) val (eval val))))
    (setq vals (get cntxtp parm))
    (if (setq pair (assoc vall vals))
       (rplaca (cdr pair) (cfcombine (cadr pair) newcf))
       (push (list vall newcf) vals))
    (storeparmvals cntxtp parm vals) ))

; EMYCIN CF calculation algorithm.
; "It ain't perfect, but it's better than its inputs usually are."
(defun cfcombine (old new)
  (cond ((null old) new)
        ((null new) old)
        ((not (and (numberp old) (numberp new)))
         (error "Bad CF combination: ~A ~A" old new))
        ((or (and (= old 1.0) (= new -1.0))
             (and (= old -1.0) (= new 1.0)))
         (error "Contradiction in CF values: ~A ~A" old new))
       ((or (= old 1.0) (= new 1.0)) 1.0)
       ((or (= old -1.0) (= new -1.0)) -1.0)
       ((and (minusp old) (minusp new))
         (- (cfcombine (- old) (- new))))
       ((and (not (minusp old)) (not (minusp new)))
         (+ old (* new (- 1.0 old))))
       (t (/ (+ old new) (- 1.0 (min (abs old) (abs new)))))) )

; Print conclusion for a parameter
(defun printconclusion (cntxt parm)
  (terpri) (print parm) (princ ":")
  (mapc #'(lambda (x)
            (cond ((and (numberp (cadr x)) (> (cadr x) 0.2))
                   (format t "  ~A (~4,2F)" (car x) (cadr x)))
                  ((and (eq (car x) 'yes) (numberp (cadr x))
                        (< (cadr x) -0.2))
                   (format t "  ~A (~4,2F)" 'no (- (cadr x))))))
        (cntxtget cntxt parm))
  (terpri) )

; Do a set of things in consequent
(defmacro do-all (&rest lvarfordoall)
   '(mapc (function eval) (quote ,lvarfordoall) ))

;----------  Utility Functions for Printing and Analysis  -------

; Show a rule in Lisp form, e.g., (showrule 'rule101).
(defun showrule (rulename) (pprint (get rulename 'rule)))

; Show properties of a context, e.g., (showprops 'rock3)
(defun showprops (atm) (pprint (cons atm (symbol-plist atm))))

; The following code translates rules to English, answers Why.
```

```
; Uses CNTXTNAME, the last context given to defcontext.
; Print out a rule in stylized English
(defun englrule (rulename)
  (prog ((rule (get rulename 'rule)) conse)
    (unless rule (return nil))
    (terpri) (terpri) (princ "If:") (terpri)
    (printpremises (cadr rule) 0 5 0)
    (terpri) (princ "then:") (terpri)
    (setq conse (caddr rule))
    (if (eq (car conse) 'conclude)
        (printconc conse)
        (mapc #'printconc (cdr conse)))
    (terpri)
    (return rulename) ))

;  Print premises of a rule
(defun printpremises (clause numb spaces initspaces)
  (prog (junction)
    (cond ((null clause)(return nil))
          ((not (member (car clause) '($and $or)))
           (printclause clause) (return nil)))
    (setq junction (car clause))
 lp (unless (setq clause (cdr clause)) (return nil))
    (spaces (- spaces initspaces))
    (setq initspaces 0)
    (setq numb (1+ numb))
    (if (< numb 10) (princ " "))
    (princ numb) (princ ") ")
    (printpremises (car clause) 0 (+ spaces 5) (+ spaces 4))
    (when (cdr clause) (princ ", ")
          (princ (case junction ($and "and") ($or "or")
                                (t junction)))
          (terpri))
    (go lp) ))

(defun spaces (n) (dotimes (i n) (princ " ")) )

(defun printclause (clause)
(let ((pred (car clause)) pair)
  (cond ((member pred '(same notsame thoughtnot))
         (cond ((eq (car (cdddr clause)) 'yes)
                (princ (clauseparm clause))
                (princ (if (eq pred 'same) " is " " is not "))
                (princ "true of the ") (princ cntxtname))
               (t (princ "the ") (princ (clauseparm clause))
                  (princ " of the ") (princ cntxtname)
                  (princ (if (eq pred 'same) " is " " is not "))
                  (princ (car (cdddr clause))) )))
        ((setq pair
           (assoc pred '((greaterp* "greater than ")
                         (greateq* "greater than or equal to ")
                         (lessp* "less than")
                         (lesseq* "less than or equal to")
                         (between* "between "))))
```

```
        (princ "the ") (princ (clauseparm clause))
        (princ " of the ") (princ cntxtname) (princ " is ")
        (princ (cadr pair)) (princ (caddr clause))
        (when (eq pred 'between*) (princ " and ")
              (princ (car (cdddr clause)))))
       ((eq pred 'known) (princ "the ")
        (princ (clauseparm clause)) (princ " of the ")
        (princ cntxtname) (princ " is known")
        (when (cdddr clause) (princ " to be ")
                  (princ (car (cdddr clause))))))  ))

;  Return the parameter name for a clause
(defun clauseparm (clause)
  (case (car clause)
        ((same notsame known thoughtnot) (caddr clause))
       ((greaterp* greateq* lessp* lesseq* between*)
          (caddr (cadr clause)))
       (t (caddr clause)))  )

(defun printconc (conc)
  (prog (tally)
    (unless (eq (car conc) 'conclude) (return nil))
    (setq tally (caddr (cdddr conc)))
    (if (and (numberp tally) (>= (abs tally) 1000))
       (princ "it is definite")
        (progn (princ "      there is")
               (princ (cond ((not (numberp tally)) " ")
                            ((< (abs tally) 500) " weakly ")
                            ((>= (abs tally) 800) " strongly ")
                            (t " ")))
               (princ "suggestive evidence")))
    (princ " (") (princ (if (numberp tally)
                            (/ (float tally) 1000.0) tally))
    (princ ")") (terpri) (princ "      that the ")
    (cond ((eq (car (cdddr conc)) 'yes) (princ cntxtname)
           (princ (if (and (numberp tally) (< tally 0))
                      " is not " " is "))
           (princ (caddr conc)) )
          (t (princ (caddr conc)) (princ " of the ")
             (princ cntxtname)
             (princ (if (and (numberp tally) (< tally 0))
                        " is not " " is "))
             (princ (car (cdddr conc))) ))
    (terpri) ))

; (why cntxt parm value) asks why a value was concluded.
; value is optional; if unspecified, it defaults to the value
; concluded most strongly.   Example:  (why rock34 identity)
(defmacro why (&optional cntxt parm value)
  '(whynotexpr (quote ,cntxt) (quote ,parm) (quote ,value) t))

; (whynot cntxt parm value) asks why a value was not concluded.
; Example:  (whynot rock34 identity coal)
(defmacro whynot (cntxt &optional parm value)
```

```
  '(whynotexpr (quote ,cntxt) (quote ,parm) (quote ,value) nil))
(defun whynotexpr (cntxt parm value whyflg)
  (prog (rules rulename rule tally concpart vals ante pair topc)
    (if (null value)
        (cond (parm (setq value parm) (setq parm cntxt)
                    (setq cntxt topctx))
              (t (setq value cntxt) (setq cntxt topctx))))
    (unless (setq topc (get cntxt 'isa))
            (princ cntxt) (princ " ??") (return nil))
    (unless parm (setq parm (car (get topc 'goals))))
    (if (and whyflg (null value))
        (setq value (vallexpr cntxt parm)))
    (setq rules (get parm 'rules))
 lp (unless (setq rulename (pop rules)) (return cntxt))
    (setq rule (get rulename 'rule))
; See if this rule concluded the value of interest
    (setq concpart (caddr rule))
    (setq vals (mapcan #'(lambda (conc)
                           (if (and (eq (car conc) 'conclude)
                                    (eq (caddr conc) parm))
                             (list (list (car (cdddr conc))
                                         (caddr (cdddr conc))))))
                       (if (eq (car concpart) 'conclude)
                           (list concpart)
                           (cdr concpart))))
    (unless (setq pair (assoc value vals)) (go lp))
; Evaluate the precondition and see if it worked.
    (unless whyflg (setq *prescan* t))
    (setq tally (tmycin-eval (cadr rule)))
    (setq *prescan* nil)
    (if (if whyflg (not (and tally (numberp tally)))
                   (and tally (numberp tally) (> tally 0.2)))
        (go lp))
    (terpri) (terpri)
    (princ "The following rule ")
    (unless whyflg (princ "might have "))
    (princ "concluded that") (terpri)
    (princ "the ") (princ parm) (princ " of ") (princ cntxt)
    (princ " was ") (princ value)
    (if whyflg (format t " (~4,2F)"
                 (* tally (/ (float (cadr pair)) 1000.0)) ))
    (terpri) (terpri)
    (showrule rulename)
    (if whyflg (go lp))
    (terpri)
    (princ "but it failed on the following condition:")
    (terpri) (terpri)
    (setq ante (cadr rule))
    (unless (eq (car ante) '$and) (pprint ante) (go lp))
lpb (unless (setq ante (cdr ante)) (go lp))
    (setq *prescan* t)
    (setq tally (tmycin-eval (car ante)))
    (setq *prescan* nil)
    (if (and tally (numberp tally) (> tally 0.2)) (go lpb))
```

```
; This clause failed.  Print it and its result.
    (pprint (car ante))
    (princ "returned CF value was ") (princ tally) (terpri)
    (go lp) ))

; The following functions were written by Hiow-Tong Jason See

;   List out all the parmeters defined in the current context.
(defun listparms ()
  (format t
    "~%~%~%--- PARAMETER LISTING OF ~a KNOWLEDGE BASE --- ~%~%"
    cntxtname)
  (dolist (x (get cntxtname 'parameters))  (format t
      "~a~%  TRANS    : ~a~%  EXPECT   : ~a~%  PROMPT : ~a~%~%"
      (first x) (printtran x) (second x) (third x))))

(defun printtran (parm)
  (if (second parm) (format nil "the ~a of * is" (car parm))
                    (format nil "~a is true of *" (car parm))) )

;  List out all the rules, in a nice format.
(defun listrules ()
  (format t "~%~%~%--- RULE LISTING OF ~a KNOWLEDGE BASE --- ~%"
            cntxtname)
  (dolist (x allrules)
    (format t "~%~%~%~a" x)
    (englrule x)
    (format t "Premise :~%")
    (pprint (cadr (get x 'rule)))
    (format t "Action  :~%")
    (pprint (caddr (get x 'rule))) ))

;   List out parameters, showing rules that might derive them.
(defun analyze-kb ()
  (format t "--- ANALYSIS OF ~a KNOWLEDGE BASE --- ~%" cntxtname)
  (format t "~%Number of Rules in the KB      : ~d"
    (length allrules))
  (format t "~%Number of Parameters in the KB : ~d"
    (length (get cntxtname 'parameters)))
  (format t "~%Initial Data needed by the KB  : ~a"
   (get cntxtname 'initialdata))
  (format t "~%Goals to be concluded          : ~a"
   (get cntxtname 'goals))
  (format t " ~%~%~%Deriviation of Parameters:~%")
  (dolist (x (get cntxtname 'parameters))
    (format t "~%~a~%   " (first x))
    (cond ((get (first x) 'rules)
           (format t "Derived from : ")
           (pprint (get (first x) 'rules)))
          (t (format t "Need input from user.~% " x)))))

;   List out everything in the Expert Knowledge Base.
(defun listkb ()
  (analyze-kb)
```

```
  (listparms)
  (listrules))

;   Clear all previous rules in the knowledge base.
(defun clear-rules ()
  (dolist (p (get cntxtname 'parameters))
          (setf (get (car p) 'rules) nil))
  (setf allrules nil))

; Delete a single rule
(defun delrule (rulename)
  (dolist (p (get cntxtname 'parameters))
    (if (member rulename (get (car p) 'rules))
        (setf (get (car p) 'rules)
              (remove rulename (get (car p) 'rules)))))
  (setf allrules (remove rulename allrules))
  rulename)
```

```
; tmycin.arl            Hiow-Tong See          2 August 1988
; Abbreviated Rule Language for Tmycin
;
; Copyright (c) 1988 by Hiow-Tong See.
; This program may be freely copied, used, or modified,
; provided that this copyright notice is included in each
; copy of the code or parts thereof.

(defun briefrule (rulename)
  (prog (rule premise conclusion)
    (or (setq rule (get rulename 'rule)) (return nil))
    (setq premise (second rule))
    (setq conclusion (third rule))
    (format t "~%Premise :~%")
    (arlpremise premise 0 2 0)
    (format t "~%Action  :~%")
    (if (eq (car conclusion) 'do-all)
      (mapc #'arlconclusion (cdr conclusion))
      (arlconclusion conclusion))))

(defun arlpremise (clause numb sp initspaces)
  (prog (junction)
    (cond ((null clause) (return nil))
      ((not (member (car clause) '($and $or)))
       (arlclause clause) (return nil)))
    (setq junction (car clause))
 lp (unless (setq clause (cdr clause)) (return nil))
    (spaces (- sp initspaces))
    (setq initspaces 0)
    (setq numb (1+ numb))
    (if (< numb 10) (format t " "))
    (format t "~A) " numb)
    (arlpremise (car clause) 0 (+ sp 5) (+ sp 4))
    (when (cdr clause)(format t "~A~%"
           (case junction ($and ",") ($or " or") (t junction))))
    (go lp)))

(defun arlclause (clause)
  (let ((pred (car clause))
        (ynparm (eq (fourth clause) 'yes)) symbol sign)
    (setq sign (if (member pred '(notsame thoughtnot notknown))
                   " ~" " "))
    (cond ((member pred '(same notsame thoughtnot))
           (if ynparm (format t "~A~A" sign (clauseparm clause))
             (format t " ~A~A= ~A" (clauseparm clause) sign
               (fourth clause))))
          ((setq symbol
             (second (assoc pred '((greaterp* ">") (greateq* ">=")
                                   (lessp* "<") (lesseq* "<="))))
           (format t " ~A ~A ~A" (clauseparm clause) symbol
             (third clause)))
          ((eq pred 'between*)
           (format t " ~A <= ~A <= ~A" (third clause)
             (clauseparm clause) (fourth clause)))
```

```
        ((member pred '(known notknown))
         (format t " ~A is~Aknown" (clauseparm clause) sign)
         (when (cdddr clause) (format t " to be ~A"
                                (fourth clause))))))))

(defun arlconclusion (conc)
  (prog (tally sign)
    (unless (eq (car conc) 'conclude)
      (return (format t "   Execute ~A~%" conc)))
    (setq tally (sixth conc))
    (setq sign (if (< tally 0) " ~" " "))
    (if (eq (fourth conc) 'yes)
      (format t "  ~A~A" sign (third conc))
      (format t "   ~A~A= ~A" (third conc) sign (fourth conc)))
    (format t " (~D)~%" (/ (float tally) 1000))))

(defun listbriefrules ()
  (mapc #'(lambda (rule) (format t "~%~A" rule)
           (briefrule rule)) allrules))
```

```
;  gclisp.aux     Gordon Novak & Bob Causey   15 August 88

;  Auxiliary functions for Golden Common Lisp

; Copyright (c) 1988 by Gordon S. Novak Jr. and Robert L. Causey
; This program may be freely copied, used, or modified,
; provided that this copyright notice is included in each
; copy of the code or parts thereof.

; These functions are designed to allow our exercise programs
; to run under Golden Common Lisp Ver. 1.1, Small Memory.
; They do not always implement the full functionality of the
; corresponding Common Lisp functions.

(unless (and (string-equal (lisp-implementation-type)
                           "GOLDEN COMMON LISP")
             (string-equal (lisp-implementation-version)
                           "1.1, Small Memory"))
        (print
 "Gclisp.aux is for use in Gold Hill Lisp, Ver 1.1, Small Memory"
           ) )

; Approximate Common Lisp gentemp
(defun gentemp (&optional prefix)
  (intern (symbol-name (gensym (or prefix "T")))) )

; Intersection of sets
(defun intersection (x y)
  (mapcan #'(lambda (item) (if (member item y) (list item)))
          x) )

; Union of sets.  Uses :test #'equal.
(defun union (x y)
  (prog ()
lp  (cond ((null x) (return y))
          ((member (car x) y :test #'equal))
          (t (setq y (cons (car x) y))))
    (setq x (cdr x))
    (go lp)))

; Partial implementation of elt, for lists.
(defun elt (seq n)
  (if (and (integerp n) (>= n 0))
      (progn (dotimes (i n) (setq seq (cdr seq))) (car seq))
      (error "Bad index to elt ~A" n)) )

; Partial implementation of the position function, for lists.
; Uses :test #'equal
(defun position (item seq)
  (do ( (idx 0 (1+ idx))
        (lst seq (cdr lst)) )
    ( (or (null lst) (equal item (car lst)) )
      (if lst idx))))
```

```
; Bubble sort: not efficient, but okay for small data sets.
(defun sort (seq pred)
  (prog (ptr done tmp)
top (setq done t)
    (setq ptr seq)
lp  (cond ((null (cdr ptr)) (if done (return seq) (go top)))
          ((funcall pred (cadr ptr) (car ptr))
           (setq done nil)
           (setq tmp (car ptr))
           (rplaca ptr (cadr ptr))
           (rplaca (cdr ptr) tmp)))
    (setq ptr (cdr ptr))
    (go lp) ))

; Error message for defsetf.
(defmacro defsetf (access-fn update-fn &rest rest)
'(format t "~%Sorry, there is no defsetf.~%"))

(defun cadddr (x) (cadr (cddr x)))

; NOTE: In order to run TMYCIN under Golden Common Lisp, it is also
; necessary to edit the functions printconclusion and whynotexpr to
; change the formats ~4,2F to be ~A .  GC Lisp does not support ~F.
```

```
;  rocks.lsp          Gordon Novak       06 June 88
;  TMYCIN Example: very small expert system to identify rocks

; Copyright (c) 1988 by Gordon S. Novak Jr.
; This program may be freely copied, used, or modified,
; provided that this copyright notice is included in each
; copy of the code or parts thereof.

(load "tmycin.lsp")                                  ; Load the system

(defcontext 'rock                                    ; Top context
            '((color (brown black white))            ; Parameters
              (hardness posnumb)
              (environment (igneous metamorphic sedimentary)
                 "What is the type of geologic environment?")
              (identity atom)
              (pretty nil                ; use nil for yes/no parms
                "What an average person would consider pretty"))
            '(color)                     ; Initialdata parameters
            '(identity))                 ; Goals

(defrules
(rule101 ($and (same cntxt color black)
               (notsame cntxt pretty yes)
               ($or (same cntxt hardness 4)
                    (same cntxt environment sedimentary)))
         (conclude cntxt identity coal tally 400))

(rule102 ($and (same cntxt color black)
               (between* (val1 cntxt hardness) 5 7)
               (notsame cntxt environment sedimentary))
          (conclude cntxt identity obsidian tally 700))

(rule103 ($and (same cntxt color white)
               (same cntxt environment sedimentary))
         (do-all (print "looks like limestone to me")
                 (conclude cntxt identity limestone tally 800)))
)
```

```
;  snakes.lsp          Gordon Novak       15 June 88

;  TMYCIN Example: expert system to identify snakes
;                  of Travis County, Texas

; Copyright (c) 1988 by Gordon S. Novak Jr.
; This program may be freely copied, used, or modified,
; provided that this copyright notice is included in each
; copy of the code or parts thereof.

; Ref: Alan Tennant, "A Field Guide to Texas Snakes",
;      Texas Monthly Press, 1985.

; "W. C. Fields liked to tell people he always kept some whiskey
;  handy in case he saw a snake - which he also kept handy."

;(load "tmycin.lsp")                    ; Load the system first

(defcontext 'snake
 '((color (tan brown black grey green pink red yellow orange)
          ("List the colors of the snake.  If there are"
           "multiple colors, use the following format:"
           "((color 1.0) ...), for example,"
           "((RED 1.0) (YELLOW 1.0) (BLACK 1.0))"))
   (size (tiny small medium large)
          ("Give the approximate size of the snake."
           "tiny = 11 inches or less, small = 12-18 inches,"
           "medium = 19-30 inches, large = over 30 inches."))
   (thickness (thin medium heavy)
          ("Compared to other snakes, is this snake"
           "quite thin, medium, or heavy-bodied?"))
   (pattern (bands stripes blotches diamonds spots speckles
                   solid)
          ("What pattern(s) are seen on the snake?"
           "stripes  = one or more stripes running lengthwise"
           "bands    = multiple bands around the body"
           "blotches = large contrasting blotches on the back"
           "spots    = small, roughly circular spots"
           "speckles = non-circular contrasting spots"
           "solid    = solid color without pattern"
           "If multiple patterns are observed, enter all"
           "using the format ((feature 1.0) ...)"))
   (features (ring-around-neck upturned-nose black-head
              black-tail)
          ("Enter features observed about snake,"
           "or UNK if unknown or absent"))
   (rattles nil "Does the snake have rattles on its tail?")
   (triangular-head nil
          ("Does the snake have a triangular head,"
           "noticeably larger than its neck?"))
   (red-and-yellow nil "Are red and yellow bands adjacent (touching)?")
   (cottonmouth nil ("Does the snake display an open mouth"
                     "that is white in color?"))
   (swims-head-out nil ("Does the snake swim with its head"
```

```
                        "held above the water?"))
    (bitten nil "Has someone been bitten by this snake?")
    (fang-marks nil
          "Are two noticeably larger fang marks visible?")
    (identity (plains-blind-snake prairie-ringneck
      western-diamondback plains-blackhead
      texas-coral-snake mexican-milk-snake))
    (environment (near-water in-water under-leaves grass woods)
          "What is the environment where the snake was seen?")
    (behavior (aggressive playing-dead)
          "Enter any unusual behaviors observed,"
          "or UNK if none or unknown.")
    (poisonous nil)
    (latin-name string)
    )
  '(color size pattern)
  '(identity latin-name poisonous)
)

(defrules

;---------------------------------------------------------------
; First we have a lot of rules to conclude exact identities
; based on good data, assuming it is available.
;---------------------------------------------------------------

(rule01 ($and (same cntxt color pink)
              (same cntxt size tiny)
              (same cntxt environment under-leaves))
        (conclude cntxt identity plains-blind-snake tally 400))

(rule01a (same cntxt identity plains-blind-snake)
         (conclude cntxt latin-name
                   "Leptotyphlops dulcis dulcis" tally 1000))

(rule05 ($and ($or (same cntxt color grey)
                   (same cntxt color brown))
              (same cntxt size small)
              (same cntxt features ring-around-neck))
         (conclude cntxt identity prairie-ringneck tally 800))

(rule05a (same cntxt identity prairie-ringneck)
         (conclude cntxt latin-name
                   "Diadophis punctatus arnyi" tally 1000))

(rule07 ($and (same cntxt color tan)
              (same cntxt size tiny)
              (same cntxt environment under-leaves))
        (conclude cntxt identity flathead-snake tally 400))

(rule07a (same cntxt identity flathead-snake)
         (conclude cntxt latin-name
                   "Tantilla gracilis" tally 1000))
```

```
(rule08 ($and (same cntxt color tan)
              (same cntxt size small)
              (same cntxt features black-head))
        (conclude cntxt identity plains-blackhead-snake
                  tally 800))

(rule08a (same cntxt identity plains-blackhead-snake)
         (conclude cntxt latin-name
                   "Tantilla nigriceps fumiceps" tally 1000))

(rule13 ($and (same cntxt color brown)
              ($or (same cntxt size tiny)
                   (same cntxt size small))
              (same cntxt pattern stripes)
              (same cntxt pattern speckles))
        (conclude cntxt identity texas-brown-snake tally 800))

(rule13a (same cntxt identity texas-brown-snake)
         (conclude cntxt latin-name
                   "Storeria dekayi texana" tally 1000))

(rule16 ($and (same cntxt color tan)
              (same cntxt size tiny)
              (same cntxt pattern stripes)
              (same cntxt pattern speckles))
        (conclude cntxt identity texas-lined-snake tally 500))

(rule16a (same cntxt identity texas-lined-snake)
         (conclude cntxt latin-name
                   "Tropidoclonion lineatum texanum"
                   tally 1000))

(rule19 ($and (same cntxt color tan)
              (same cntxt color white)
              ($or (same cntxt size tiny)
                   (same cntxt size small))
              (same cntxt pattern bands))
        (conclude cntxt identity ground-snake tally 600))

(rule19a (same cntxt identity ground-snake)
         (conclude cntxt latin-name
                   "Sonora semiannulata" tally 1000))

(rule20 ($and (same cntxt color pink)
              (same cntxt size tiny)
              (same cntxt environment grass))
        (conclude cntxt identity western-smooth-earth-snake
                  tally 400))

(rule20a (same cntxt identity western-smooth-earth-snake)
         (conclude cntxt latin-name
                   "Virginia valeriae elegans" tally 1000))

(rule21 ($and (same cntxt color brown)
```

```
            (same cntxt size tiny))
        (conclude cntxt identity rough-earth-snake
                  tally 300))

(rule21a (same cntxt identity rough-earth-snake)
         (conclude cntxt latin-name
                   "Virginia striatula" tally 1000))

(rule50 ($and ($or (same cntxt color tan)
                   (same cntxt color yellow))
              (same cntxt color brown)
              ($or (same cntxt pattern blotches)
                   (same cntxt pattern diamonds))
              (same cntxt size large)
              (notsame cntxt triangular-head yes)
              (notsame cntxt rattles yes))
        (conclude cntxt identity bullsnake tally 600))

(rule50a (same cntxt identity bullsnake)
         (conclude cntxt latin-name
                   "Pituophis melanoleucus sayi" tally 1000))

(rule70 ($and (same cntxt color black)
              (same cntxt size large))
        (conclude cntxt identity texas-indigo-snake
                  tally 600))

(rule70a (same cntxt identity texas-indigo-snake)
         (conclude cntxt latin-name
                   "Drymarchon corais erebennus" tally 1000))

(rule71 ($and (same cntxt color brown)
              ($or (same cntxt size medium)
                   (same cntxt size small))
              ($or (same cntxt environment in-water)
                   (same cntxt environment near-water))
              (notsame cntxt thickness heavy)
              (notsame cntxt cottonmouth yes)
              (notsame cntxt swims-head-out yes))
        (conclude cntxt identity blotched-water-snake tally 400))

(rule71a (same cntxt identity blotched-water-snake)
         (conclude cntxt latin-name
                   "Nerodia erythrogaster transversa" tally 1000))

(rule90 ($and (same cntxt color red)
              (same cntxt color yellow)
              (same cntxt color black)
              (notsame cntxt red-and-yellow yes))
         (conclude cntxt identity mexican-milk-snake tally 800))

(rule90a (same cntxt identity mexican-milk-snake )
         (conclude cntxt latin-name
                   "Lampropeltis triangulum annulata" tally 1000))
```

```
(rule96 ($and (same cntxt color red)
              (same cntxt color yellow)
              (same cntxt color black)
              (same cntxt red-and-yellow yes))
         (conclude cntxt identity texas-coral-snake tally 1000))

(rule96a (same cntxt identity texas-coral-snake)
         (conclude cntxt latin-name
                   "Micrurus fulvius tenere" tally 1000))

(rule97 ($and (same cntxt color brown)
              ($or (same cntxt size medium)
                   (same cntxt size large))
              ($or (same cntxt environment in-water)
                   (same cntxt environment near-water))
              ($or (same cntxt thickness heavy)
                   (same cntxt cottonmouth yes)
                   (same cntxt swims-head-out yes)))
        (conclude cntxt identity water-moccasin tally 600))

(rule97a (same cntxt identity water-moccasin)
         (conclude cntxt latin-name
                   "Agkistrodon piscivorus leucostoma" tally 1000))

(rule99 ($and (same cntxt color brown)
              (same cntxt color tan)
              ($or (same cntxt pattern bands)
                   (same cntxt pattern blotches))
              ($or (same cntxt size small)
                   (same cntxt size medium))
              (same cntxt environment woods)
              (same cntxt triangular-head yes))
        (conclude cntxt identity copperhead tally 800))

(rule99a (same cntxt identity copperhead)
         (conclude cntxt latin-name
                   "Agkistrodon contortrix laticinctus"
                   tally 1000))

(rule101 ($and ($or (same cntxt size medium)
                    (same cntxt size large))
               (same cntxt color brown)
               ($or (same cntxt pattern diamonds)
                    (same cntxt pattern blotches))
               ($or (same cntxt rattles yes)
                    (same cntxt triangular-head yes)))
         (conclude cntxt identity western-diamondback tally 800))

(rule101a (same cntxt identity western-diamondback)
         (conclude cntxt latin-name
                   "Crotalus atrox" tally 1000))

(rule300 ($or (same cntxt identity western-diamondback)
              (same cntxt identity rattlesnake)
```

```
            (same cntxt identity texas-coral-snake)
            (same cntxt identity water-moccasin)
            (same cntxt identity copperhead))
        (conclude cntxt poisonous yes tally 1000))

;---------------------------------------------------------------
; In some cases, good data is not available, but we would still
; like to conclude some useful information.
;---------------------------------------------------------------

; None of the striped or spotted snakes in this area are poisonous.
(rule401 ($or (same cntxt pattern stripes)
              (same cntxt pattern spots))
         (conclude cntxt poisonous yes tally -600))

(rule402 ($and (notsame cntxt identity western-diamondback)
               (notsame cntxt identity texas-coral-snake)
               (notsame cntxt identity water-moccasin)
               (notsame cntxt identity copperhead)
               (notsame cntxt identity rattlesnake))
         (conclude cntxt poisonous yes tally -300))

(rule403 ($and (same cntxt rattles yes)
               (notsame cntxt size large))
         (conclude cntxt identity rattlesnake tally 800))

(rule403a ($and (same cntxt identity rattlesnake)
                (same cntxt triangular-head yes))
          (conclude cntxt latin-name
                    "Crotalus" tally 1000))

(rule403b ($and (same cntxt identity rattlesnake)
                (thoughtnot cntxt triangular-head yes))
          (conclude cntxt latin-name
                    "Sistrurus" tally 1000))

(rule404 ($and (same cntxt bitten yes)
               (same cntxt fang-marks yes))
         (conclude cntxt poisonous yes tally 201))

(rule405 ($and (same cntxt bitten yes)
               (notsame cntxt fang-marks yes))
         (conclude cntxt poisonous yes tally -300))
)
```