TMYCIN Expert System Tool

Gordon S. Novak Jr.
Department of Computer Sciences
University of Texas at Austin
Austin, TX 78712
novak@cs.utexas.edu

1 Introduction

TMYCIN ("Tiny EMYCIN") is a simple expert system tool patterned after the EMYCIN tool developed at Stanford [1] [2]. 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 parameter
            '(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 optional <prompt> 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 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 $CF \leq .2$; 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; this 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 `CONCLUDE`d 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)))
```

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>)` will delete the single rule `<rulename>`.

`(clear-rules)` will delete all rules.[1]

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))
```

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

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)
```

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

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.

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.

References

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

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