Outline Proposal for a Prolog 'Textual Tree Tracer' (TTT)

Chris Taylor, Benedict du Boulay and Mukesh J. Patel
School of Cognitive and Computing Sciences
University of Sussex

10th January 1991

Abstract

This document constitutes an outline specification for a new tracer for Prolog, the design of which has been guided by an evaluation of the relative strengths and weaknesses of a number of existing Prolog tracers. The new tracer, known as the 'Textual Tree Tracer' (or 'TTT' for short) will produce a 'sideways tree' representation of the execution of a goal, using only textual output, i.e. it will not require the use of any specialised graphics. Its key features include the following: a compact and yet very informative basic form of output, which distinguishes clause matching events, and several different goal failure modes; clear display of the structure of computation and the flow of control, via the use of a tree representation; extensive use of default controls to limit the quantity of trace output produced; the facility of retrospective inspection of earlier parts of the trace, in order to obtain more detailed information; and a specialised 'database window' which facilitates correlation of the trace with the source code, and shows dynamically any changes to the database resulting from the assertion or retraction of clauses.

Contents

1 Background

This proposal arises in the context of a research project currently in progress at the University of Sussex, entitled "Explanation Facilities for Prolog: Towards More Versatile Intelligent Tutoring Systems", which is funded by the Joint Research Council's "Human-Computer Interaction Initiative". One of the stated aims of that project is the re-implementation of a Prolog tracer called the "Extended Prolog Tracer for Beginners" (EPTB), written in prototype form by Christo Dichev and Benedict du Boulay (see [Dichev & du Boulay 89]). However, the project has also involved the investigation of other existing tracers, notably the "Transparent Prolog Machine" (TPM), a graphical tracer produced originally by Eisenstadt and Brayshaw, currently at the Open University (see e.g. [Eisenstadt & Brayshaw 88]), and the standard 'Byrd box' or 'spy' tracer (see [Byrd 80] and [Clocksin & Mellish 81]). Experience in using the EPTB tracer and a commercially available version of the TPM tracer has suggested that the original aim of re-implementing the EPTB should be replaced by the aim of implementing a new tracer which incorporates good features from both the TPM and the EPTB, whilst overcoming some of their deficiences. Accordingly, this report constitutes an outline specification for such a tracer.

2 Tracer design considerations

2.1 General tracer types

In this section, some informal criteria are suggested for describing Prolog tracing tools.

Textual/graphical. A *textual* tracer is one whose output consists only of alphanumeric and other keyboard characters; whilst a *graphical* tracer is one which makes use also of graphical devices such as boxes, circles, and connecting lines, and which requires the use of a graphics workstation. Textual tracers include the spy, the EPTB, the APT (Rajan86) and the PTP (Eisenstadt84). The latter two of these, whilst not graphical in the full sense, use visual techniques such as highlighting to augment certain aspects of the textual display. Graphical tracers are exemplified by the various versions of the TPM. See also DewarANDCleary86 for an account of an earlier tracer using graphical techniques.

Linear/non-linear. A tracer is *linear* if its output proceeds in a simple sequential manner, with each new item of the trace immediately following the previous one (apart from any 'white space' or punctuation characters which may intervene) in the display window — usually in a left to right, top to bottom order. By contrast, a *non-linear* tracer is one in which the cursor may flit around the screen, adding new symbols in the middle of the trace output as currently displayed, as well as at its end. The spy and the EPTB are examples of linear tracers, whereas the TPM is an example of a non-linear tracer.

Indented/unindented. A tracer is *indented* if it makes use of indentation from an edge of the display window — usually the left-hand or top edge — to encode information about the computational depth of goals; otherwise, it is *unindented*. The PTP and the TPM use indentation, whilst the EPTB and the spy do not.

2.2 Design principles

As with design in general, the design of a Prolog tracing tool ought to be preceded by an attempt to formulate some design principles. A number of such principles are now proposed. For an earlier set of design principles similar in some respects to these, see for example [Rajan 90].

Use of meaningful symbolism. Tracer output should make use of symbolism whose meaning is readily apparent, or easy to comprehend, e.g. by using mnemonic abbreviations: the use of totally abstract symbolism should be avoided as far as possible.

Correlation of trace with source code. The tracer should be designed so as to enhance the ease with which the trace output can be correlated with the source code of the program being traced.

Purity of trace. The trace output should not incorporate anything which is not part of the trace itself, such as for example commands input by the user, or listings of procedures from the programs being traced.

Inviolability of the program. It should not be possible for the program being traced to be altered, or effectively altered, by the tracing tool, in the middle of a trace, since to permit this would be to encourage undisciplined program development methods. The phrase 'effectively altered' here covers the use of such tracer options as 'forced failure', in which a goal that would actually succeed during the unfettered execution of a program is made to fail artificially by a command input to the tracer by the user.

Compactness of information displayed. Any information represented in the trace should be encoded as compactly as it can be without loss of clarity and readability.

Default restriction of output. The tracer should be constructed so that unless given specific instructions to the contrary, it produces only a limited amount of basic information, regarded as the most important: any additional information should be available on request, but should not be part of the basic trace output.

Preservation of trace information. As the trace proceeds, information that was displayed at an earlier stage in the trace — for example previous variable bindings — should never be irretrievably lost. This need not mean that such information should be displayed throughout the generation of the trace: however, it should at least be accessible on request from the user.

Encoding of structure of computation. The trace should encode explicitly the structure of the computation, i.e. it should represent both parent-child relations between goal nodes, and also the order of execution of sibling goal nodes. The most effective way to do this is to for the output to use a 'tree' representation of some kind.

3 Comparison of three existing tracers

In this section, attention will be confined to the following three tracers: the EPTB, a version of the TPM, and the standard 'Byrd box' or 'spy' tracer that is provided with most implementations of Prolog. No detailed description of these tracers will be given here — the reader should refer to the references given already in section 1.

The following tables, whilst not claimed to be completely comprehensive, provide an overview of the threetracers and their relative merits. The evaluative judgements made reflect only the subjective opinions of the authors, rather than being based on any wider survey of opinion. The entries on the TPM refer to an implementation of it marketed by Chemical Design Ltd. — henceforth referred to as CDL-TPM — which is inferior to the Open University's 1987 Apollo implementation, as that is described in the literature. This latter version, which can be conveniently referred to as JLP-TPM, since it is virtually identical to the system described in the Journal of Logic Programming (EisenstadtANDBrayshaw88, is an improvement on CDL-TPM in at least the following respects: it can handle endless loops; shows actual variable names and clause instantiations; and shows variable bindings explicitly. Nonetheless, some of the weaknesses of the TPM — such as the inconvenient display of and access to variable bindings and the arguments of goals — are arguably inherent in its basic design (which makes use of a vertically aligned tree, so that the horizontal space available between nodes is limited).

3.1 General tracer type

	Spy	EPTB	CDL-TPM
Textual/graphical	Textual	Textual	Graphical
Linear/non-linear	Linear	Linear	Non-linear
Indented/unindented	Unindented	Unindented	Indented

3.2 Explicit information provided

Information either shown automatically, or on request.

	Spy	EPTB	CDL-TPM
Identifying nos. of matching clauses	No	Yes	Yes
Instantiated instances of clauses	No	Yes	No
Different goal failure modes distinguished	No	Yes	Yes
Ancestors of a goal	No	Yes	Yes
Descendants of a goal	No	Yes	Yes
System goals visible	No	No	Yes
Actual variable names shown in trace	No	Yes	No
Variable bindings shown explicitly	No	Yes	No

3.3 Facilities and options

	Spy	EPTB	CDL-TPM
Trapping a particular named procedure	Yes	Yes	Yes
Trapping goals matching a template	No	Yes	Yes
Reverse stepping through trace	No	No	Yes
Forced failure of goals available	Yes	No	No
Unleashing available	Yes	Yes	Yes

3.4 Adherence to design principles

	Spy	EPTB	CDL-TPM
Use of meaningful symbolism	Fair	Fair	Fair
Correlation of trace with source code	Fair	Fair	Poor
Purity of trace	No	No	Yes
Inviolability of program	No	Yes	Yes
Compactness of information displayed	Fair	Poor	Fair(2)
Default restriction of output	Poor	Poor	Fair
Preservation of trace information	Yes	Yes	Yes
Encoding of structure of computation	Poor	Fair(1)	Good

Notes:

1. Rather than using indentation, the EPTB uses numerical labels to indicate the computational depth of goals. These labels are not shown by default, but are included in the trace if the user so requests.

2. The TPM (in all versions) provides a compact 'long-distance view' of the proof tree corresponding to the evaluation of a goal (see [Eisenstadt & Brayshaw 88]). However, since this 'long-distance view' contains very little detailed information, it is usually necessary for the user to request expanded views of parts of the tree, displayed in separate windows; and these expanded views use a more informative but considerably less compact tree representation.

3.5 Some other characteristics

	Spy	EPTB	CDL-TPM
Display of backtracking	Poor	Poor	Good
Accessibility of variable bindings	Poor	Good	Poor(1)
User-friendliness of command interface	Fair	Poor	Fair
Range of information and options available	Poor	Good	Poor
Clarity of trace layout	Fair	Poor	Fair
Location of loops in programs	Fair	Fair	Very poor(2)

Notes:

1. In the CDL-TPM, the variable bindings associated with a goal are not immediately visible — to access them, the user has to select the corresponding goal node in the 'long distance view' of the proof tree, using a mouse-controlled cursor. A more detailed view of that node, showing the arguments of the goal, is then displayed in a small window. However, only a few such windows can be displayed simultaneously, and in some cases, such windows may need to be scrolled in order to see all of the arguments. Furthermore, like the 'spy', the CDL-TPM does not show the actual variable names used in clauses or top-level goals. This is in contrast to the EPTB tracer and the JLP-TPM, which do show the actual variable names.

2. The CDL-TPM does not display a proof tree for a goal until the attempt to satisfy the goal has terminated (either successfully or unsuccessfully). Thus if a program contains a loop, no trace output is produced at all, and so the user receives no indications as to the location of the loop. This problem has been overcome in the JLP-TPM, in which a 'live mode' is

available in which display of the proof tree proceeds concurrently with its development.

To conclude this section, the nature of the three tracers can be summarised as follows. The 'Byrd box' or 'spy' tracer provides a fairly compact and simple trace output, from which the bindings of variables in goals can be inferred, although not very conveniently. However, it gives a poor picture of the structure of the computation, and provides no explicit information about which clauses are being tried, or (at least, in the version available in the Poplog system) about system calls.

By comparison, the EPTB tracer provides much more information, but at the expense of a simple interface, and a compact trace output. Like the spy, it does not show system goals, and it fails to show the structure of the computation clearly, since it is a linear and unindented tracer.

The CDL-TPM shows the structure of the computation and the flow of control very clearly, and like the EPTB provides information concerning clause matching and modes of failure, but is weak as regards the convenient display and access of variable bindings and the arguments of goals. The JPL-TPM implementation is an improvement in this respect, in that it shows variable bindings explicitly and uses actual variable names: however, it still has to rely on a limited number of sub-windows for the display of goal arguments and variable bindings, so that goal terms and their associated arguments and variable bindings are not automatically visible, and only a limited number of them can be seen at any one time when compared to the more compact of the textual tracers.

4 Specification of an improved tracer

4.1 General concept

In section 2, a number of strengths and weaknesses have been identified for the EPTB, CDL-TPM and Byrd box tracers. Consideration of these has led to the conception of a new tracer which, it is hoped, will combine the best features from these earlier tracers (along with some other features), whilst at the same time avoiding their various shortcomings.

The new tracer will be called the 'Textual Tree Tracer' (TTT for short): that is to say, it will output a representation of a proof tree, as does the CDL-TPM tracer; but it will do so using textual output, rather than specialised graphics, and it will make available detailed information about variable bindings, on request, as does the EPTB. As the examples to follow will illustrate, the TTT will use a 'sideways tree', with the 'root node' at the top left, with branches growing towards the right, and with new subtrees of a node being added below any previous subtrees of that node. This is in contrast to the TPM, which uses an upside-down vertically aligned tree, with the root node near the top of the screen.

The TTT will make use of the following windows: the 'trace window' — showing the trace output itself; the 'database window' — showing the current state of the program database; the 'help window' — showing information concerning the commands available to the user at any particular point, the history of commands entered by the user, and certain other helpful information; the 'trace window' — showing the trace output itself; and the 'input/output window' (or windows), showing any program input and output. These windows could either be Ved windows in the Poplog system (on a terminal without graphics facilities), or — on a Sun workstation — Sunview windows or Xwindows. (N.B. 'Ved' is an editor used in the Poplog system.) The 'help window' and the 'input/output window' will appear only intermittently, whilst the 'trace window' and the 'database window' will be visible for most or all of the time. The software controlling the windows will be written using abstract (i.e. system independent) window manipulation procedures, so that it can subsequently be adapted quite easily to use Ved, Sunview, Xwindows, or some other system, by defining the abstract procedures in terms of the appropriate system-specific procedures.

4.1.1 Database window.

The 'database window' will be particularly useful for debugging programs involving the assertion and retraction of clauses, but more generally, it will help the user to relate the trace output to the program being traced. Of course, any window displaying the source code would help to do this, but the 'database window' will do this in an enhanced way by explicitly numbering the clauses of procedures, and by indenting the subgoals of clauses in a way which closely resembles the way in which calls to subgoals will be shown in the trace output. In this respect, a textual tracer using a sideways tree offers considerable advantages over a graphical tracer using a vertically aligned tree.

A 'database window' was used earlier in Rajan's APT tracer [Rajan 86], and a reference to the general concept can be found in [Eisenstadt, Hasemer & Kriwaczek]. In the TTT (unlike the APT), there will be few dynamic changes to the appearance of the 'database window', apart from the display of any assertions and retractions of clauses. One useful feature, however, might be the automatic dynamic adjustment of the database window so as to keep visible within the window the procedure corresponding to the latest goal under evaluation (since unless the program is a trivial one, the program clauses will typically take up too much space for them all to be visible in the database window at once). Whether or not such automatic adjustment is performed, it ought to be possible for the user to scan the database clauses at any point during the trace, either by using Ved keys — if a Ved editor window is being used — or by scrolling.

4.1.2 Help window.

The help window will show dynamically the sequence of commands issued by the user during the execution of the trace. On request from the user, it will also show the command options currently available to the user (this will vary according to the circumstances), and indicate which of those options is the current default option. In addition, the help window will be used to display other useful information, such as explanations of the meanings of certain symbols used in the trace output.

4.1.3 Trace window.

This window will contain the trace output itself. As already described, it will be in the form of a sideways tree, with its root at the top left of the trace, making use of indentation from the left-hand edge to encode the depth of goal nodes in the proof tree. In this respect, the TTT tracer will produce a tree quite similar to that produced by the 'Mellish tracer' available in the Poplog system: however, unlike that tracer, it will never irretrievably overwrite any information displayed earlier in the production of the trace; and in addition, its output will be much more compact than that of the Mellish tracer, even though it will provide considerably more information (although not all of this information will be shown by default).

4.1.4 Input/output window or windows.

In order to conform to the design principle of 'purity of the trace', any output produced by the program being traced, and any input to it, will not be shown in the middle of the trace, but in a separate input/output (I/O) window or windows. If a program has more than one I/O device or file, the I/O window could be subdivided into different sub-windows — one for each I/O device or file — or an entirely separate window could be used for each I/O device or file.

4.1.5 Control of the tracer.

A key feature of the I/O interface of the TTT tracer will be its sensitivity to the position of a mobile cursor in the 'trace window', moved about either with the 'arrow' keys, or — if this is available — under mouse control. The range of trace control options available to the user will depend upon the current cursor position, and — in some cases — on the most recent previous command or commands, as stored on a 'command stack' maintained by the tracer. On request from the user, the 'help window' will be used to display the current command stack and the command options currently available to the user. One of the commands available will be the default in those circumstances (executed by simply hitting the 'return' key), whilst the others will require more specific inputs.

At any point during the development of the trace, the user will be able to move the cursor around over the current trace output; otherwise, when the cursor is not being controlled explicitly by the user, its movement will be determined automatically by the tracer.

Commands will be input to the tracer typically by typing alphanumeric codes. Extensive use will be made of default commands, which will be executed by hitting the 'return' key, or in some cases, as a result of the movement of the cursor.

The commands that will be available can be classified into a few basic groups, as follows:

1. 'Retrospective display' commands.
   Commands which allow the user to modify the information displayed in that part of the trace output which has been produced so far. These commands could be further subdivided into: (i) Goal visibility commands — controlling the visibility or otherwise of goals in the trace; and (ii) Goal detail commands — controlling the amount of detail shown about goals which are currently visible. (N.B. Retrospective techniques are also used in the PTP and TPM tracers — see [Eisenstadt 84] and [Eisenstadt & Brayshaw 88], respectively.)

2. 'Prospective display' commands.
   Commands which alter the information that will be displayed in the trace from the current point in the computation of the trace onwards. Again, these commands may effect either the visibility of goals, or the amount of information shown in connection with those goals that will be made visible. (N.B. It will not be necessary to issue any such commands at the start of the trace, because the trace will begin with certain default controls on the information displayed.)

3. 'Help' commands.
   Commands which cause helpful information to be displayed in the 'help window' — for example explanations of symbols used in the trace window, or lists of commands available in the current context (as determined by the current cursor location and stack of previous commands).

4. 'Trace driving' commands.
   Commands which enable the trace to be developed further forwards, or to be 'unravelled' backwards to an earlier point in the computation. Usually the trace will be developed or unravelled in a stepwise mode, i.e. it will stop subsequently for further input from the user: however, there will also be commands analogous to the 'unleash' command in the 'spy' tracer — which causes the trace to move onwards without any further interruption for inputs from the user. When operating in stepwise mode, the next stopping point for input will depend upon the current controls on the information to be made visible.

5. 'Window dump' commands.
   Commands which cause the current contents of any of the tracer windows to be saved to a file (either a default file, or some other specified file), or to be sent directly to a printer to obtain a hard copy. Some such commands might already be available as part of the underlying system (i.e. Xwindows, Sunview, etc.) in which the tracer program is operating.

In general, for any command which can be 'undone', there will be a corresponding 'inverse' command, e.g. for any command which adds some information to that currently on display in the trace window, there will be a corresponding inverse command which removes that information from the display.

One of the distinctive features of the TTT tracer will be the extensive use of default controls aimed at restricting the amount of information produced in the trace output. This will be achieved in part by default restrictions on the predicates shown in the trace (see Example 10), and on the level of detail shown concerning each goal (see Examples 1 – 9 for illustrations of the minimum amount of information shown about individual goals). It will also be achieved by the default execution, under many circumstances, of commands which remove from display any detailed information requested earlier by the user. Such commands would be executed automatically by movement of the cursor away from the cursor location from which the display of the information was requested: in order to retain such information in the display, the user would have to input explicit 'retention' commands. Extra information could be displayed at an intermediate intensity level — so as to stand out against the rest of the trace — until made permanent by the execution of a retention command, when it would be displayed at the normal intensity level. The same technique of using intermediate intensity might also be used to indicate branches of the proof tree which have failed.

No attempt will be made here to decide precisely what commands should be available under various circumstances, or to decide which of them should be defaults: it would be unwise to try to specify this in detail in advance, since practical experience of using a prototype is required to arrive at a sensible selection of commands and defaults. However, examples 12 to 16 inclusive (in section 4.6) indicate some possible commands that may prove to be useful.

4.2 Underlying structure of the trace

The tracing of a goal in the new tracer will involve the construction of an AND-OR proof tree, represented by a Prolog term, whose abstract structure is quite similar to that of the trees constructed by the TPM tracer, although some extra features will be added.

As with the TPM, trees will be composed of 'goal nodes' and 'clause nodes'. Most goal nodes will have one or more clause nodes as their immediate child nodes — the exceptions being goal nodes for system goals, which will have no child nodes, and goal nodes for ';' constructions (such as "(p(X);q(X))" and top-level compound goals (as in ?- p(X), q(X).), which will have other goal nodes as their child nodes. Unless it corresponds to a clause with no subgoals, any clause node will have one or more goal nodes as its child nodes.

A subtree of the proof tree with a goal node as its root will be represented by a Prolog term of the following (approximate) form:

```
goal_node(
    Goal_number,
    Initial_goal_term,  /* Goal term as instantiated on calling */
    Current_goal_term,  /* Goal term as currently instantiated */
    Goal status field,  /* Indicates whether goal has succeeded,
                           failed, or is being evaluated etc. */
    Variable_binding_field,  /* Contains information about
                                bindings of variables in goal */
    Node_visibility_flag,    /* Indicates if goal is currently
                                shown in trace output */
    Cursor_location_flag,    /* Indicates if cursor is currently
                                located on this goal */
    ....   possibly other arguments
    Subtree_list)         /* N.B. This list is empty if the node
                                   is a leaf node. */
```

A subtree with a clause node as its root will be represented by a Prolog term of the following (approximate) form:

```
clause_node(
    Predicate_name,
    Predicate_arity,
```

```
Absolute_clause_no,
Current_ordinal_clause_no, /* Indicates current ordinal
                              position of clause in database */
Clause_before_matching_goal,
Clause_after_matching_goal,
Clause_as_currently_instantiated,
Clause_variable_binding_field,  /* Contains information about
                                   clause variable bindings */
Node_visibility_flag,
Cursor_location_flag,
....   possibly other arguments
Subtree_list)
```

4.3 Basic symbolism used

The following symbols and abbreviations will be used in the TTT output (their use will become clearer in the examples of trace output given later — this section should be used primarily for reference):

```
***1:, **23:, *124:  etc.   Goal number labels (nos. correspond to
                            nos. of calls in 'spy' trace).
$$$
N.B. The use of leading asterisks here means that each goal number label
occupies the same amount of space - 5 characters including the colon
(allowing up to 9999 goals in any single trace). This helps to emphasise
the indentation pattern of the goals. Also, for those goals numbered less
than 1000 (i.e.\ most of the goals encountered in practice), the asterisks
help to make the lines representing goals stand out clearly against the
rest of the trace.

1, 2, 10 etc.               Clause number labels (used in
                            'database window', as well as in
                            the actual trace).

| and / are used as punctuation characters.

=   is used to indicate current variable bindings.
```

- # is used to indicate variable bindings which are no longer current (see Examples 3 & 4), and also - in the 'database window' - clauses which have been retracted from the database (see example 19).

- ! is used in the goal status fields of goals which have been 'cutoff' by the action of the 'cut' (see Example 6).

- , is used in the goal status field of compound goals using ',' (such as ''?- p(X), q(X)'') to separate the goal status fields of the individual goals.

- ; is used in the goal status field of disjunctive compound goals (such as ''(p(X) ; q(X))'' to separate the goal status fields of the individual disjuncts.

- ? is used to indicate goals and clauses currently being evaluated (including re-evaluation on backtracking).

- S Indicates success of a goal. Repeated successes on backtracking are represented by a sequence of S characters - one for each success, e.g. SSS means that the goal so labelled has succeeded three times.

Failure modes (as in the EPTB, TPM and PTP, several failure modes are distinguished):

- Fm 'match failure'. Clauses of same name and arity as goal are in the database, but none of them match the goal.

- Fa 'arity failure'. No clauses of same name and arity as goal are in the database, but some clause or clauses of the same name and different arity are in the database.

- Fu 'undefined procedure failure'. No clauses of the same name as the goal (of whatever arity) are in the database.

- Fc 'cutoff failure'. Failure due to attempting to resatisfy a goal which has been 'cut off' by the action of the 'cut'.

`Fb` ‘backtracking failure’. Failure on backtracking of a goal which initially succeeded, either because no more clauses are left which match the goal, or because the goal is a system goal which cannot be resatisfied.

`Fs` ‘subgoal failure’. Failure resulting from failure of one of the subgoals of a clause.

`F` Any other failure - includes ‘first-time’ (i.e. not on backtracking) failure of system goals, and failure of compound disjuncts of disjunctive goals, i.e. disjuncts consisting of more than one goal.

4.4 Basic trace output

As explained in section 4.1.4, default curbs on output will be in operation most of the time, so that most goals, and any additional detailed information about goals or clauses, will not be shown automatically unless specific commands to the contrary are given by the user. An illustration of the action of the default curbs will be given in section 4.5: however, in this section, for the purposes of illustrating the symbolism used in the basic trace output, it will be assumed that the user has requested that all goals should be shown in the trace.

Example 1: Flow of control.

This example shows the step-by-step evolution of the TTT trace, with the corresponding ‘spy’ trace for comparison. It demonstrates how backtracking and clause retrying are dealt with. In order to avoid distracting detail at this stage, the program used involves only goals without arguments.

Points to note:

1. The TTT trace is very compact — in this example it takes up only about one third of the space used by the corresponding spy trace. This compactness is achieved because the success, failure and redoing of calls are represented by suffixes to the lines representing the calls, rather than by extra lines in the trace (like the ‘Exit :’, ‘Fail :’ and ‘Redo :’ lines in the spy trace). However, if there are variables involved, extra lines are needed to display their bindings, so in general, unless backtracking occurs, the TTT trace will be similar in length to the corresponding spy trace.

2. In spite of its compactness, the TTT trace — even in its basic form — provides considerably more information than the corresponding spy trace, because the TTT shows the numbers of matching clauses, and distinguishes a greater number of failure modes than the spy does.

3. Unlike the spy trace, the TTT trace shows clearly the structure of the computation, by making use of indentation.

4. By means of the non-sequential movement of the cursor, the TTT shows very clearly the flow of control involved in backtracking. In this respect it is like the TPM — another non-linear tracer — and unlike the spy or the EPTB, which are linear tracers.

```
Program clauses as shown in 'database window':

1   s.

1   t.

1   p:-
     q,
     r.

1   q:-
     s.
2   q:-
     t.

Goal: ?- p.

Development of the TTT trace, compared with spy trace:

(N.B. It is assumed that the user just keeps hitting the 'return'
key. Note that the TTT trace involves more key presses than the
spy, so that successive stages of the spy trace - shown on the
left-hand side - are sometimes identical.

Spy trace                                            TTT trace

** (1) Call : p?                                     ***1: p  ?

** (1) Call : p?                                     ***1: p  1?
```

```
** (1) Call : p?                                       ***1: p  1?
** (2) Call : q?                                        ***2: q  ?

** (1) Call : p?                                       ***1: p  1?
** (2) Call : q?                                        ***2: q  1?

** (1) Call : p?                                       ***1: p  1?
** (2) Call : q?                                        ***2: q  1?
** (3) Call : s?                                         ***3: s  ?

** (1) Call : p?                                       ***1: p  1?
** (2) Call : q?                                        ***2: q  1?
** (3) Call : s?                                         ***3: s  1?

** (1) Call : p?                                       ***1: p  1?
** (2) Call : q?                                        ***2: q  1?
** (3) Call : s?                                         ***3: s  1S
** (3) Exit : s?

** (1) Call : p?                                       ***1: p  1?
** (2) Call : q?                                        ***2: q  1S
** (3) Call : s?                                         ***3: s  1S
** (3) Exit : s?
** (2) Exit : q?

** (1) Call : p?                                       ***1: p  1?
** (2) Call : q?                                        ***2: q  1S
** (3) Call : s?                                         ***3: s  1S
** (3) Exit : s?                                        ***4: r  ?
** (2) Exit : q?
** (4) Call : prolog_error(UNDEFINED PREDICATE [r])?

** (1) Call : p?                                       ***1: p  1?
** (2) Call : q?                                        ***2: q  1S
** (3) Call : s?                                         ***3: s  1S
** (3) Exit : s?                                        ***4: r  Fu
** (2) Exit : q?
** (4) Call : prolog_error(UNDEFINED PREDICATE [r])?
** (4) Fail : prolog_error(UNDEFINED PREDICATE [r])?
```

```
** (1) Call : p?                                              ***1: p  1?
** (2) Call : q?                                               ***2: q  1S?
** (3) Call : s?                                                ***3: s  1S
** (3) Exit : s?                                               ***4: r  Fu
** (2) Exit : q?
** (4) Call : prolog_error(UNDEFINED PREDICATE [r])?
** (4) Fail : prolog_error(UNDEFINED PREDICATE [r])?
** (2) Redo : q?

** (1) Call : p?                                              ***1: p  1?
** (2) Call : q?                                               ***2: q  1S?
** (3) Call : s?                                                ***3: s  1S?
** (3) Exit : s?                                               ***4: r  Fu
** (2) Exit : q?
** (4) Call : prolog_error(UNDEFINED PREDICATE [r])?
** (4) Fail : prolog_error(UNDEFINED PREDICATE [r])?
** (2) Redo : q?
** (3) Redo : s?

** (1) Call : p?                                              ***1: p  1?
** (2) Call : q?                                               ***2: q  1S?
** (3) Call : s?                                                ***3: s  1SFb
** (3) Exit : s?                                               ***4: r  Fu
** (2) Exit : q?
** (4) Call : prolog_error(UNDEFINED PREDICATE [r])?
** (4) Fail : prolog_error(UNDEFINED PREDICATE [r])?
** (2) Redo : q?
** (3) Redo : s?
** (3) Fail : s?

** (1) Call : p?                                              ***1: p  1?
** (2) Call : q?                                               ***2: q  1SFb/2?
** (3) Call : s?                                                ***3: s  1SFb
** (3) Exit : s?                                               ***4: r  Fu
** (2) Exit : q?
** (4) Call : prolog_error(UNDEFINED PREDICATE [r])?
** (4) Fail : prolog_error(UNDEFINED PREDICATE [r])?
** (2) Redo : q?
** (3) Redo : s?
** (3) Fail : s?

** (1) Call : p?                                              ***1: p  1?
** (2) Call : q?                                               ***2: q  1SFb/2?
** (3) Call : s?                                                ***3: s  1SFb
```

```
** (3) Exit : s?                                       ***5: t  ?
** (2) Exit : q?                                      ***4: r  Fu
** (4) Call : prolog_error(UNDEFINED PREDICATE [r])?
** (4) Fail : prolog_error(UNDEFINED PREDICATE [r])?
** (2) Redo : q?
** (3) Redo : s?
** (3) Fail : s?
** (5) Call : t?

** (1) Call : p?                                       ***1: p  1?
** (2) Call : q?                                        ***2: q  1SFb/2?
** (3) Call : s?                                         ***3: s  1SFb
** (3) Exit : s?                                         ***5: t  1?
** (2) Exit : q?                                        ***4: r  Fu
** (4) Call : prolog_error(UNDEFINED PREDICATE [r])?
** (4) Fail : prolog_error(UNDEFINED PREDICATE [r])?
** (2) Redo : q?
** (3) Redo : s?
** (3) Fail : s?
** (5) Call : t?

** (1) Call : p?                                     ***1: p  1?
** (2) Call : q?                                      ***2: q  1SFb/2?
** (3) Call : s?                                       ***3: s  1SFb
** (3) Exit : s?                                       ***5: t  1S
** (2) Exit : q?                                      ***4: r  Fu
** (4) Call : prolog_error(UNDEFINED PREDICATE [r])?
** (4) Fail : prolog_error(UNDEFINED PREDICATE [r])?
** (2) Redo : q?
** (3) Redo : s?
** (3) Fail : s?
** (5) Call : t?
** (5) Exit : t?

** (1) Call : p?                                     ***1: p  1?
** (2) Call : q?                                      ***2: q  1SFb/2S
** (3) Call : s?                                       ***3: s  1SFb
** (3) Exit : s?                                       ***5: t  1S
** (2) Exit : q?                                      ***4: r  Fu
** (4) Call : prolog_error(UNDEFINED PREDICATE [r])?
** (4) Fail : prolog_error(UNDEFINED PREDICATE [r])?
** (2) Redo : q?
** (3) Redo : s?
** (3) Fail : s?
```

```
** (5) Call : t?
** (5) Exit : t?
** (2) Exit : q?

** (1) Call : p?                                    ***1: p  1?
** (2) Call : q?                                     ***2: q  1SFb/2S
** (3) Call : s?                                      ***3: s  1SFb
** (3) Exit : s?                                      ***5: t  1S
** (2) Exit : q?                                     ***4: r  Fu
** (4) Call : prolog_error(UNDEFINED PREDICATE [r])?  ***6: r  ?
** (4) Fail : prolog_error(UNDEFINED PREDICATE [r])?
** (2) Redo : q?
** (3) Redo : s?
** (3) Fail : s?
** (5) Call : t?
** (5) Exit : t?
** (2) Exit : q?
** (6) Call : prolog_error(UNDEFINED PREDICATE [r])?

** (1) Call : p?                                    ***1: p  1?
** (2) Call : q?                                     ***2: q  1SFb/2S
** (3) Call : s?                                      ***3: s  1SFb
** (3) Exit : s?                                      ***5: t  1S
** (2) Exit : q?                                     ***4: r  Fu
** (4) Call : prolog_error(UNDEFINED PREDICATE [r])?  ***6: r  Fu
** (4) Fail : prolog_error(UNDEFINED PREDICATE [r])?
** (2) Redo : q?
** (3) Redo : s?
** (3) Fail : s?
** (5) Call : t?
** (5) Exit : t?
** (2) Exit : q?
** (6) Call : prolog_error(UNDEFINED PREDICATE [r])?
** (6) Fail : prolog_error(UNDEFINED PREDICATE [r])?

** (1) Call : p?                                    ***1: p  1?
** (2) Call : q?                                     ***2: q  1SFb/2S?
** (3) Call : s?                                      ***3: s  1SFb
** (3) Exit : s?                                      ***5: t  1S
** (2) Exit : q?                                     ***4: r  Fu
** (4) Call : prolog_error(UNDEFINED PREDICATE [r])?  ***6: r  Fu
** (4) Fail : prolog_error(UNDEFINED PREDICATE [r])?
** (2) Redo : q?
** (3) Redo : s?
```

```
** (3) Fail : s?
** (5) Call : t?
** (5) Exit : t?
** (2) Exit : q?
** (6) Call : prolog_error(UNDEFINED PREDICATE [r])?
** (6) Fail : prolog_error(UNDEFINED PREDICATE [r])?
** (2) Redo : q?

** (1) Call : p?                                     ***1: p  1?
** (2) Call : q?                                      ***2: q  1SFb/2S?
** (3) Call : s?                                       ***3: s  1SFb
** (3) Exit : s?                                       ***5: t  1S?
** (2) Exit : q?                                      ***4: r  Fu
** (4) Call : prolog_error(UNDEFINED PREDICATE [r])?   ***6: r  Fu
** (4) Fail : prolog_error(UNDEFINED PREDICATE [r])?
** (2) Redo : q?
** (3) Redo : s?
** (3) Fail : s?
** (5) Call : t?
** (5) Exit : t?
** (2) Exit : q?
** (6) Call : prolog_error(UNDEFINED PREDICATE [r])?
** (6) Fail : prolog_error(UNDEFINED PREDICATE [r])?
** (2) Redo : q?
** (5) Redo : t?

** (1) Call : p?                                     ***1: p  1?
** (2) Call : q?                                      ***2: q  1SFb/2S?
** (3) Call : s?                                       ***3: s  1SFb
** (3) Exit : s?                                       ***5: t  1SFb
** (2) Exit : q?                                      ***4: r  Fu
** (4) Call : prolog_error(UNDEFINED PREDICATE [r])?   ***6: r  Fu
** (4) Fail : prolog_error(UNDEFINED PREDICATE [r])?
** (2) Redo : q?
** (3) Redo : s?
** (3) Fail : s?
** (5) Call : t?
** (5) Exit : t?
** (2) Exit : q?
** (6) Call : prolog_error(UNDEFINED PREDICATE [r])?
** (6) Fail : prolog_error(UNDEFINED PREDICATE [r])?
** (2) Redo : q?
** (5) Redo : t?
** (5) Fail : t?
```

```
** (1) Call : p?                                     ***1: p  1?
** (2) Call : q?                                      ***2: q  1SFb/2SFb
** (3) Call : s?                                       ***3: s  1SFb
** (3) Exit : s?                                       ***5: t  1SFb
** (2) Exit : q?                                      ***4: r  Fu
** (4) Call : prolog_error(UNDEFINED PREDICATE [r])?  ***6: r  Fu
** (4) Fail : prolog_error(UNDEFINED PREDICATE [r])?
** (2) Redo : q?
** (3) Redo : s?
** (3) Fail : s?
** (5) Call : t?
** (5) Exit : t?
** (2) Exit : q?
** (6) Call : prolog_error(UNDEFINED PREDICATE [r])?
** (6) Fail : prolog_error(UNDEFINED PREDICATE [r])?
** (2) Redo : q?
** (5) Redo : t?
** (5) Fail : t?
** (2) Fail : q?

** (1) Call : p?                                     ***1: p  1Fs
** (2) Call : q?                                      ***2: q  1SFb/2SFb
** (3) Call : s?                                       ***3: s  1SFb
** (3) Exit : s?                                       ***5: t  1SFb
** (2) Exit : q?                                      ***4: r  Fu
** (4) Call : prolog_error(UNDEFINED PREDICATE [r])?  ***6: r  Fu
** (4) Fail : prolog_error(UNDEFINED PREDICATE [r])?  no
** (2) Redo : q?
** (3) Redo : s?
** (3) Fail : s?
** (5) Call : t?
** (5) Exit : t?
** (2) Exit : q?
** (6) Call : prolog_error(UNDEFINED PREDICATE [r])?
** (6) Fail : prolog_error(UNDEFINED PREDICATE [r])?
** (2) Redo : q?
** (5) Redo : t?
** (5) Fail : t?
** (2) Fail : q?
** (1) Fail : p?
no
```

Example 2: Goal variable bindings.

Points to note:

1. The basic trace output will show only the bindings of variables which are unbound when goals are called — hence it will not invariably show the bindings of clause variables.

2. In the basic trace, only the bindings resulting from success of goals will be shown, i.e. any intermediate bindings will not be shown. However, as will be described in section 4.4, more information about variable bindings will be available on request.

3. Only the numbers of clauses whose heads match goals appear in the trace. For example, since '2' is the first number to appear at the right-hand side of the first line of the trace, it can be inferred that clause 1 of 'conc' did not match the goal 'conc([a, b], [c], L)'.

4. Like the EPTB, and unlike the spy or TPM, the TTT will show the actual names of variables. The following method for displaying names will be used: top-level goal variables will be shown just as they are in the goal (e.g. 'L' in this example), whereas clause variable names are shown as they occur in the clauses of the program, but with a numerical suffix added (e.g. 'T1_1', 'T1_2' in this example) so that variables associated with different invocations of a clause can be distinguished from one another.

5. The number of the clause that was used to obtain a set of bindings is indicated at the left-hand side of the line showing the first binding in that set.

```
Program as shown in 'database window':

1  conc([], L, L).
2  conc([H|T], L, [H|T1]):-
     conc(T, L, T1).

Goal: ?- conc([a,b], [c], L).

Development of the TTT trace:

***1: conc([a,b], [c], L)  ?

***1: conc([a,b], [c], L)  2?

***1: conc([a,b], [c], L)  2?
 ***2: conc([b], [c], T1_1)  ?
```

```
***1: conc([a,b], [c], L)  2?
 ***2: conc([b], [c], T1_1)  2?

***1: conc([a,b], [c], L)  2?
 ***2: conc([b], [c], T1_1)  2?
  ***3: conc([], [c], T1_2)  ?

***1: conc([a,b], [c], L)  2?
 ***2: conc([b], [c], T1_1)  2?
  ***3: conc([], [c], T1_2)  1?

***1: conc([a,b], [c], L)  2?
 ***2: conc([b], [c], T1_1)  2?
  ***3: conc([], [c], T1_2)  1?
  |1    T1_2 = [c]

***1: conc([a,b], [c], L)  2?
 ***2: conc([b], [c], T1_1)  2?
  ***3: conc([], [c], T1_2)  1S
  |1    T1_2 = [c]

***1: conc([a,b], [c], L)  2?
 ***2: conc([b], [c], T1_1)  2?
 |2    T1_1 = [b,c]
  ***3: conc([], [c], T1_2)  1S
  |1    T1_2 = [c]

***1: conc([a,b], [c], L)  2?
 ***2: conc([b], [c], T1_1)  2S
 |2    T1_1 = [b,c]
  ***3: conc([], [c], T1_2)   1S
  |1    T1_2 = [c]

***1: conc([a,b], [c], L)  2?
|2    L = [a,b,c]
 ***2: conc([b], [c], T1_1)  2S
 |2    T1_1 = [b,c]
  ***3: conc([], [c], T1_2)  1S
  |1    T1_2 = [c]
```

```
***1: conc([a,b], [c], L)  2S
|2    L = [a,b,c]
 ***2: conc([b], [c], T1_1)  2S
 |2    T1_1 = [b,c]
  ***3: conc([], [c], T1_2)  1S
  |1    T1_2 = [c]

***1: conc([a,b], [c], L)  2S
|2    L = [a,b,c]
 ***2: conc([b], [c], T1_1)  2S
 |2    T1_1 = [b,c]
  ***3: conc([], [c], T1_2)  1S
  |1    T1_2 = [c]
yes
```

Example 3: Variable unbinding on backtracking.

Points to note:

1. When a variable becomes unbound as a result of backtracking, the old binding is still shown, but the "=" sign is replaced by a hash character (#). Any new bindings are shown on fresh lines below it.

```
Program as shown in 'database window' of tracer:

1  p(X):-
    q(X),
    r(X)
2  p(c)

1  q(a)
2  q(b)

1  r(d)

Goal: ?- p(A).

Development of the TTT trace:

***1: p(A)  ?

***1: p(A)  1?
```

```
***1: p(A)  1?
 ***2: q(A)  ?

***1: p(A)  1?
 ***2: q(A)  1?

***1: p(A)  1?
 ***2: q(A)  1?
 |1    A = a

***1: p(A)  1?
 ***2: q(A)  1S
 |1    A = a

***1: p(A)  1?
 ***2: q(A)  1S
 |1    A = a
 ***3: r(a)  ?

***1: p(A)  1?
 ***2: q(A)  1S
 |1    A = a
 ***3: r(a)  Fm

***1: p(A)  1?
 ***2: q(A)  1S?
 |1    A # a
 ***3: r(a)  Fm

***1: p(A)  1?
 ***2: q(A)  1SFb/2?
 |1    A # a
 ***3: r(a)  Fm

***1: p(A)  1?
 ***2: q(A)  1SFb/2?
 |1    A # a
 |2    A = b
 ***3: r(a)  Fm

***1: p(A)  1?
 ***2: q(A)  1SFb/2S
 |1    A # a
 |2    A = b
```

```
 ***3: r(a)  Fm

***1: p(A)  1?
 ***2: q(A)  1SFb/2S
 |1    A # a
 |2    A = b
 ***3: r(a)  Fm
 ***4: r(b)  ?

***1: p(A)  1?
 ***2: q(A)  1SFb/2S
 |1    A # a
 |2    A = b
 ***3: r(a)  Fm
 ***4: r(b)  Fm

***1: p(A)  1?
 ***2: q(A)  1SFb/2S?
 |1    A # a
 |2    A # b
 ***3: r(a)  Fm
 ***4: r(b)  Fm

***1: p(A)  1?
 ***2: q(A)  1SFb/2SFb
 |1    A # a
 |2    A # b
 ***3: r(a)  Fm
 ***4: r(b)  Fm

***1: p(A)  1Fs/2?
 ***2: q(A)  1SFb/2SFb
 |1    A # a
 |2    A # b
 ***3: r(a)  Fm
 ***4: r(b)  Fm

***1: p(A)  1Fs/2?
|2    A = c
 ***2: q(A)  1SFb/2SFb
 |1    A # a
 |2    A # b
 ***3: r(a)  Fm
 ***4: r(b)  Fm
```

```
***1: p(A)  1Fs/2S
|2    A = c
 ***2: q(A)  1SFb/2SFb
 |1    A # a
 |2    A # b
 ***3: r(a)  Fm
 ***4: r(b)  Fm

***1: p(A)  1Fs/2S
|2    A = c
 ***2: q(A)  1SFb/2SFb
 |1    A # a
 |2    A # b
 ***3: r(a)  Fm
 ***4: r(b)  Fm
yes
```

Example 4: Multiple variable bindings and answers.

This example illustrates what happens when the initial query contains more than one variable, and when multiple answers to the query are obtained by backtracking at the top-level goal, forced by the user.

Points to note:

1. To save space, the X and Y bindings could be put on the same line for each individual answer. However, that would mean that the computational level of Y would not be so clearly indicated as it is by its alignment with the start of the 'p' goal, and in any case, it is not very common for a goal to have more than 1 or 2 uninstantiated variables. (Refer back to example 2 to see how variables are indented in accordance with their computational level.)

```
Program clauses as shown by 'database window':

1  q(a, b).
2  q(d, a).

1  p(e, f).
2  p(X, Y):-
     q(X, Y).

Goal: ?- p(X, Y).
```

```
Development of the TTT trace:

***1: p(X, Y)  ?

***1: p(X, Y)  1?

***1: p(X, Y)  1?
|1    X = e
|     Y = f

***1: p(X, Y)  1S
|1    X = e
|     Y = f

Now user forces backtracking by inputting ‘;’ (not shown on trace)

***1: p(X, Y)  1S?
|1    X # e
|     Y # f

***1: p(X, Y)  1SFb/2?
|1    X # e
|     Y # f

***1: p(X, Y)  1SFb/2?
|1    X # e
|     Y # f
 ***2: q(X, Y)  ?

***1: p(X, Y)  1SFb/2?
|1    X # e
|     Y # f
 ***2: q(X, Y)  1?

***1: p(X, Y)  1SFb/2?
|1    X # e
|     Y # f
 ***2: q(X, Y)  1?
 |1    X = a
 |     Y = b
```

```
***1: p(X, Y)  1SFb/2?
|1    X # e
|     Y # f
 ***2: q(X, Y)  1S
 |1    X = a
 |     Y = b

***1: p(X, Y)  1SFb/2?
|1    X # e
|     Y # f
|2    X = a
|     Y = b
 ***2: q(X, Y)  1S
 |1    X = a
 |     Y = b

***1: p(X, Y)  1SFb/2S
|1    X # e
|     Y # f
|2    X = a
|     Y = b
 ***2: q(X, Y)  1S
 |1    X = a
 |     Y = b

(Now user forces backtracking again.)

***1: p(X, Y)  1SFb/2S?
|1    X # e
|     Y # f
|2    X # a
|     Y # b
 ***2: q(X, Y)  1S
 |1    X # a
 |     Y # b
```

```
***1: p(X, Y)  1SFb/2S?
|1    X # e
|     Y # f
|2    X # a
|     Y # b
 ***2: q(X, Y)  1S?
 |1    X # a
 |     Y # b

***1: p(X, Y)  1SFb/2S?
|1    X # e
|     Y # f
|2    X # a
|     Y # b
 ***2: q(X, Y)  1SFb/2?
 |1    X # a
 |     Y # b

***1: p(X, Y)  1SFb/2S?
|1    X # e
|     Y # f
|2    X # a
|     Y # b
 ***2: q(X, Y)  1SFb/2?
 |1    X # a
 |     Y # b
 |2    X = d
 |     Y = a

***1: p(X, Y)  1SFb/2S?
|1    X # e
|     Y # f
|2    X # a
|     Y # b
 ***2: q(X, Y)  1SFb/2S
 |1    X # a
 |     Y # b
 |2    X = d
 |     Y = a
```

```
***1: p(X, Y)  1SFb/2S?
|1    X # e
|     Y # f
|2    X # a
|     Y # b
|2    X = d
|     Y = a
 ***2: q(X, Y)  1SFb/2S
 |1    X # a
 |     Y # b
 |2    X = d
 |     Y = a

***1: p(X, Y)  1SFb/2SS
|1    X # e
|     Y # f
|2    X # a
|     Y # b
|2    X = d
|     Y = a
 ***2: q(X, Y)  1SFb/2S
 |1    X # a
 |     Y # b
 |2    X = d
 |     Y = a

***1: p(X, Y)  1SFb/2SS
|1    X # e
|     Y # f
|2    X # a
|     Y # b
|2    X = d
|     Y = a
 ***2: q(X, Y)  1SFb/2S
 |1    X # a
 |     Y # b
 |2    X = d
 |     Y = a
yes
```

Example 5: System predicates and infix operators.

Only the complete trace is shown in this example.

Points to note:

1. Like the TPM, and unlike the spy and the EPTB, the TTT tracer will show system calls, thereby making it easier to correlate the trace output with the program clauses.
2. For infix goals, whether they are system goals or not, the default will be to show them in infix form (as for the ‘$<$’ in this example), although as in the EPTB, it will be possible to show them in non-infix form on request.

```
Program as shown in ‘database window’:

1  sorted([]).
2  sorted([X]):-
    integer(X).
3  sorted([X, Y|T]):-
    integer(X),
    integer(Y),
    X < Y,
    sorted([Y|T]).

Goal: ?- sorted([2,3,6]).

Complete TTT trace:

***1: sorted([2,3,6])  3S
 ***2: integer(2)  S
 ***3: integer(3)  S
 ***4: 2 < 3  S
 ***5: sorted([3,6])  3S
  ***6: integer(3)  S
  ***7: integer(6)  S
  ***8: 3 < 6  S
  ***9: sorted([6])  2S
   **10: integer(6)  S
yes
```

Example 6: Effect of the 'cut'.

Points to note:

1. Immediately after a 'cut' has succeeded, those goals which have been 'cut off' are indicated by the insertion of a '!' symbol in their goal status fields. This is analogous to the use of 'clouding' over part of the graphical proof tree, used to show the effect of the cut in the Open University's version of the TPM.

`Program clauses as shown in 'database window':`

```
1  q.

1  p:-
    q,
    !,
    r.
```

`Goal: ?- p.`

`Development of the TTT trace:`

```
***1: p  ?

***1: p  1?

***1: p  1?
 ***2: q  ?

***1: p  1?
 ***2: q  1?

***1: p  1?
 ***2: q  1S

***1: p  1?
 ***2: q  1S
 ***3: !  ?

***1: p  1?
 ***2: q  1S
 ***3: !  S
```

```
***1: p  1!?
 ***2: q  1S!
 ***3: !  S

***1: p  1!?
 ***2: q  1S!
 ***3: !  S
 ***4: r  ?

***1: p  1!?
 ***2: q  1S!
 ***3: !  S
 ***4: r  Fu

***1: p  1!?
 ***2: q  1S!
 ***3: !  S?
 ***4: r  Fu

***1: p  1!?
 ***2: q  1S!
 ***3: !  SFb
 ***4: r  Fu

***1: p  1!Fc
 ***2: q  1S!
 ***3: !  SFb
 ***4: r  Fu

***1: p  1!Fc
 ***2: q  1S!
 ***3: !  SFb
 ***4: r  Fu
no
```

Example 7: Display of deep trees.

Points to note:

1. The indentation reverts to the left-hand side of the window after a certain depth is reached (in this case a depth of 8). The depth at which this occurs will be set to a default value, although could be made alterable by the user.

```
    This example shows just the complete trace.

Program as shown in ‘database window’:

1  member(X, [X|_]).
2  member(X, [_|T]):-
    member(X, T).

Goal: ?- member(5, [7,8,0,6,3,2,34,9,100,5,11]).


Complete TTT trace:

***1: member(5, [7,8,0,6,3,2,34,9,100,5,11])  2S
 ***2: member(5, [8,0,6,3,2,34,9,100,5,11])  2S
  ***3: member(5, [0,6,3,2,34,9,100,5,11])  2S
   ***4: member(5, [6,3,2,34,9,100,5,11])  2S
    ***5: member(5, [3,2,34,9,100,5,11])  2S
     ***6: member(5, [2,34,9,100,5,11])  2S
      ***7: member(5, [34,9,100,5,11])  2S
       ***8: member(5, [9,100,5,11])  2S
 _________/
/
***9: member(5, [100,5,11])  2S
 **10: member(5, [5,11])  1S
yes
```

Example 8: Top-level conjunctive goals.

Points to note:

1. The individual goals of a top-level conjunctive goal (e.g. the goals numbered '***2' and '***3' in this example) are shown as its immediate subnodes, i.e. they are indented one character further to the right.

2. The goal status field of a conjunctive goal (such as that for the goal numbered '***1' in this example) consists of one or more subfields separated by ',' characters, each corresponding to one of the individual goals of the conjunctive goal. As the individual goals are evaluated, and their individual goal status fields develop, so the subfields of the goal status field of the conjunctive goal develop in parallel, as illustrated in this example.

```
Program clauses as shown in database window:

1  conc([H|T], L, [H|T1]):-
    conc(T, L, T1).
2  conc([], L, L).

Goal: ?- conc([], [a], X), conc(X, [b], Y), fail.


Development of the TTT trace (assuming user keeps hitting the 'return' key,
and all goals are being traced):

***1: conc([], [a], X), conc(X, [b], Y), fail  ?

***1: conc([], [a], X), conc(X, [b], Y), fail  ?
 ***2: conc([], [a], X)  ?

***1: conc([], [a], X), conc(X, [b], Y), fail  2?
 ***2: conc([], [a], X)  2?

***1: conc([], [a], X), conc(X, [b], Y), fail  2?
 ***2: conc([], [a], X)  2?
 |2    X = [a]

***1: conc([], [a], X), conc(X, [b], Y), fail  2S,?
 ***2: conc([], [a], X)  2S
 |2    X = [a]

***1: conc([], [a], X), conc(X, [b], Y), fail  2S,?
 ***2: conc([], [a], X)  2S
```

```
 |2    X = [a]
 ***3: conc([a], [b], Y)  ?

***1: conc([], [a], X), conc(X, [b], Y), fail  2S,1?
 ***2: conc([], [a], X)  2S
 |2    X = [a]
 ***3: conc([a], [b], Y)  1?

***1: conc([], [a], X), conc(X, [b], Y), fail  2S,1?
 ***2: conc([], [a], X)  2S
 |2    X = [a]
 ***3: conc([a], [b], Y)  1?
  ***4: conc([], [b], T1_1)  ?

***1: conc([], [a], X), conc(X, [b], Y), fail  2S,1?
 ***2: conc([], [a], X)  2S
 |2    X = [a]
 ***3: conc([a], [b], Y)  1?
  ***4: conc([], [b], T1_1)  2?

***1: conc([], [a], X), conc(X, [b], Y), fail  2S,1?
 ***2: conc([], [a], X)  2S
 |2    X = [a]
 ***3: conc([a], [b], Y)  1?
  ***4: conc([], [b], T1_1)  2S
  |2    T1_1 = [b]

***1: conc([], [a], X), conc(X, [b], Y), fail  2S,1?
 ***2: conc([], [a], X)  2S
 |2    X = [a]
 ***3: conc([a], [b], Y)  1?
 |1    Y = [a,b]
  ***4: conc([], [b], T1_1)  2S
  |2    T1_1 = [b]

***1: conc([], [a], X), conc(X, [b], Y), fail  2S,1S,?
 ***2: conc([], [a], X)  2S
 |2    X = [a]
 ***3: conc([a], [b], Y)  1S
 |1    Y = [a,b]
  ***4: conc([], [b], T1_1)  2S
  |2    T1_1 = [b]

***1: conc([], [a], X), conc(X, [b], Y), fail  2S,1S,?
```

```
***2: conc([], [a], X)  2S
|2    X = [a]
***3: conc([a], [b], Y)  1S
|1    Y = [a,b]
 ***4: conc([], [b], T1_1)  2S
 |2    T1_1 = [b]
***5: fail  ?

***1: conc([], [a], X), conc(X, [b], Y), fail  2S,1S,F
 ***2: conc([], [a], X)  2S
 |2    X = [a]
 ***3: conc([a], [b], Y)  1S
 |1    Y = [a,b]
  ***4: conc([], [b], T1_1)  2S
  |2    T1_1 = [b]
 ***5: fail  F

***1: conc([], [a], X), conc(X, [b], Y), fail  2S,1S?,F
 ***2: conc([], [a], X)  2S
 |2    X = [a]
 ***3: conc([a], [b], Y)  1S
 |1    Y = [a,b]
  ***4: conc([], [b], T1_1)  2S
  |2    T1_1 = [b]
 ***5: fail  F

***1: conc([], [a], X), conc(X, [b], Y), fail  2S,1S?,F
 ***2: conc([], [a], X)  2S
 |2    X = [a]
 ***3: conc([a], [b], Y)  1S?
 |1    Y # [a,b]
  ***4: conc([], [b], T1_1)  2S?
  |2    T1_1 # [b]
 ***5: fail  F

***1: conc([], [a], X), conc(X, [b], Y), fail  2S,1S?,F
 ***2: conc([], [a], X)  2S
 |2    X = [a]
 ***3: conc([a], [b], Y)  1S?
 |1    Y # [a,b]
  ***4: conc([], [b], T1_1)  2SFb
  |2    T1_1 # [b]
 ***5: fail  F
```

```
***1: conc([], [a], X), conc(X, [b], Y), fail  2S?,1SFb,F
 ***2: conc([], [a], X)  2S
 |2    X = [a]
 ***3: conc([a], [b], Y)  1SFb
 |1    Y # [a,b]
  ***4: conc([], [b], T1_1)  2SFb
  |2    T1_1 # [b]
 ***5: fail  F

***1: conc([], [a], X), conc(X, [b], Y), fail  2S?,1SFb,F
 ***2: conc([], [a], X)  2S?
 |2    X # [a]
 ***3: conc([a], [b], Y)  1SFb
 |1    Y # [a,b]
  ***4: conc([], [b], T1_1)  2SFb
  |2    T1_1 # [b]
 ***5: fail  F

***1: conc([], [a], X), conc(X, [b], Y), fail  2SFb,1SFb,F
 ***2: conc([], [a], X)  2SFb
 |2    X # [a]
 ***3: conc([a], [b], Y)  1SFb
 |1    Y # [a,b]
  ***4: conc([], [b], T1_1)  2SFb
  |2    T1_1 # [b]
 ***5: fail  F

***1: conc([], [a], X), conc(X, [b], Y), fail  2SFb,1SFb,F
 ***2: conc([], [a], X)  2SFb
 |2    X # [a]
 ***3: conc([a], [b], Y)  1SFb
 |1    Y # [a,b]
  ***4: conc([], [b], T1_1)  2SFb
  |2    T1_1 # [b]
 ***5: fail  F
no
```

Example 9: Disjunctive goals.

Points to note:

1. A disjunctive goal (such as the goal labelled '***2' in this example) has a goal status field consisting of two sub-fields — one for each disjunct — separated by a ';'. If the second disjunct is not tried, the second subfield will be blank.
2. If the first attempt to satisfy a compound disjunct (i.e. a disjunct consisting of more than one goal, like the first disjunct in this example) fails, this is indicated by an 'F' in the corresponding subfield of the goal status field for the disjunction. Failure of a disjunct on backtracking would be indicated by 'Fb'; and failure of a first attempt to satisfy a disjunct consisting of a single goal would be indicated by the code for the failure mode of that goal, e.g. 'Fm', 'Fu', or 'Fs'.

N.B. See also Example 10, which shows how variable bindings associated with disjunctive goals are displayed.

```
Program clauses as shown in database window:

1  q(a)

1  r(a)

1  t(a)

1  p(X):-
    (q(X),r(X),s(X);t(X)),
    fail.

Goal: ?- p(a).


Development of the TTT trace:

***1: p(a)  ?

***1: p(a)  1?

***1: p(a)  1?
 ***2: (q(a),r(a),s(a);t(a))  ?

***1: p(a)  1?
 ***2: (q(a),r(a),s(a);t(a))  ?;
```

```
***1: p(a)  1?
 ***2: (q(a),r(a),s(a);t(a))  ?;
  ***3: q(a)  ?

***1: p(a)  1?
 ***2: (q(a),r(a),s(a);t(a))  ?;
  ***3: q(a)  1?

***1: p(a)  1?
 ***2: (q(a),r(a),s(a);t(a))  ?;
  ***3: q(a)  1S

***1: p(a)  1?
 ***2: (q(a),r(a),s(a);t(a))  ?;
  ***3: q(a)  1S
  ***4: r(a)  ?

***1: p(a)  1?
 ***2: (q(a),r(a),s(a);t(a))  ?;
  ***3: q(a)  1S
  ***4: r(a)  1?

***1: p(a)  1?
 ***2: (q(a),r(a),s(a);t(a))  ?;
  ***3: q(a)  1S
  ***4: r(a)  1S

***1: p(a)  1?
 ***2: (q(a),r(a),s(a);t(a))  ?;
  ***3: q(a)  1S
  ***4: r(a)  1S
  ***5: s(a)  ?

***1: p(a)  1?
 ***2: (q(a),r(a),s(a);t(a))  ?;
  ***3: q(a)  1S
  ***4: r(a)  1S
  ***5: s(a)  Fu
```

```
***1: p(a)  1?
 ***2: (q(a),r(a),s(a);t(a))  ?;
  ***3: q(a)  1S
  ***4: r(a)  1S?
  ***5: s(a)  Fu

***1: p(a)  1?
 ***2: (q(a),r(a),s(a);t(a))  ?;
  ***3: q(a)  1S
  ***4: r(a)  1SFb
  ***5: s(a)  Fu

***1: p(a)  1?
 ***2: (q(a),r(a),s(a);t(a))  ?;
  ***3: q(a)  1S?
  ***4: r(a)  1SFb
  ***5: s(a)  Fu

***1: p(a)  1?
 ***2: (q(a),r(a),s(a);t(a))  ?;
  ***3: q(a)  1SFb
  ***4: r(a)  1SFb
  ***5: s(a)  Fu

***1: p(a)  1?
 ***2: (q(a),r(a),s(a);t(a))  F;?
  ***3: q(a)  1SFb
  ***4: r(a)  1SFb
  ***5: s(a)  Fu

***1: p(a)  1?
 ***2: (q(a),r(a),s(a);t(a))  F;?
  ***3: q(a)  1SFb
  ***4: r(a)  1SFb
  ***5: s(a)  Fu
  ***6: t(a)  ?

***1: p(a)  1?
 ***2: (q(a),r(a),s(a);t(a))  F;?
  ***3: q(a)  1SFb
  ***4: r(a)  1SFb
  ***5: s(a)  Fu
  ***6: t(a)  1?
```

```
***1: p(a)  1?
 ***2: (q(a),r(a),s(a);t(a))  F;?
  ***3: q(a)  1SFb
  ***4: r(a)  1SFb
  ***5: s(a)  Fu
  ***6: t(a)  1S

***1: p(a)  1?
 ***2: (q(a),r(a),s(a);t(a))  F;S
  ***3: q(a)  1SFb
  ***4: r(a)  1SFb
  ***5: s(a)  Fu
  ***6: t(a)  1S

***1: p(a)  1?
 ***2: (q(a),r(a),s(a);t(a))  F;S
  ***3: q(a)  1SFb
  ***4: r(a)  1SFb
  ***5: s(a)  Fu
  ***6: t(a)  1S
 ***7: fail  ?

***1: p(a)  1?
 ***2: (q(a),r(a),s(a);t(a))  F;S
  ***3: q(a)  1SFb
  ***4: r(a)  1SFb
  ***5: s(a)  Fu
  ***6: t(a)  1S
 ***7: fail  F

***1: p(a)  1?
 ***2: (q(a),r(a),s(a);t(a))  F;S?
  ***3: q(a)  1SFb
  ***4: r(a)  1SFb
  ***5: s(a)  Fu
  ***6: t(a)  1S
 ***7: fail  F
```

```
***1: p(a)  1?
 ***2: (q(a),r(a),s(a);t(a))  F;S?
  ***3: q(a)  1SFb
  ***4: r(a)  1SFb
  ***5: s(a)  Fu
  ***6: t(a)  1S?
 ***7: fail  F

***1: p(a)  1?
 ***2: (q(a),r(a),s(a);t(a))  F;S?
  ***3: q(a)  1SFb
  ***4: r(a)  1SFb
  ***5: s(a)  Fu
  ***6: t(a)  1SFb
 ***7: fail  F

***1: p(a)  1?
 ***2: (q(a),r(a),s(a);t(a))  F;SFb
  ***3: q(a)  1SFb
  ***4: r(a)  1SFb
  ***5: s(a)  Fu
  ***6: t(a)  1SFb
 ***7: fail  F

***1: p(a)  1Fs
 ***2: (q(a),r(a),s(a);t(a))  F;SFb
  ***3: q(a)  1SFb
  ***4: r(a)  1SFb
  ***5: s(a)  Fu
  ***6: t(a)  1SFb
 ***7: fail  F

***1: p(a)  1Fs
 ***2: (q(a),r(a),s(a);t(a))  F;SFb
  ***3: q(a)  1SFb
  ***4: r(a)  1SFb
  ***5: s(a)  Fu
  ***6: t(a)  1SFb
 ***7: fail  F
no
```

4.5 Default controls on trace output

In both the spy and Dichev tracers, it is possible to specify that tracing output should be restricted to certain predicates only. However, the default behaviour — i.e. if the tracer is just started without any additional specific control information — is to trace every predicate (except for system predicates, which are not shown by either the EPTB, or many implementations of the spy). In many cases, the practical effect of this is a huge and unwieldy trace, which makes it difficult to home in with precision on the source of any bugs that are present. (N.B. In the spy and the EPTB, the quantity of output can be reduced by the use of the 'skip' command: when applied to a goal which has just been called, it suppresses display of any subgoals called in evaluating that goal, so that only the outcome of its evaluation (success or failure, plus any bindings that result) is shown. However, the 'skip' command is not applied by default - it has to be input explicitly by the user — and so there is a tendency for it to be used less frequently than its benefits would merit.)

The TTT trace output in its basic form (illustrated earlier in section 4.3.1) is already compact — in most cases, at least as compact as the spy trace output, and in many cases, more compact, particularly when backtracking occurs. Even so, however, unrestricted tracing of predicates will still tend to generate traces of considerable size: hence, as already indicated in section 4.1.5, it is desirable to introduce some other means of restricting the output generated. This will be achieved by various defaul t controls on the information made visible in the trace. Examples 10 and 11 deal with such controls.

Example 10: Illustration of default curbs on output.

The program in this example is intended as an economical way of representing the following undirected graph:

```
a   c
 \ / \
  b   d
```

The top-level procedure 'link(X,Y)' is intended to succeed if there is a path (possibly empty, if X and Y are the same node) in the graph which connects the nodes X and Y. Points to note:

1. This example illustrates how the trace output develops if the trace is started without any specific instructions being issued by the user concerning the information to be made visible. The default behaviour under these circumstances is for the tracer to show only the basic trace output for the top-level goal itself, and its immediate sub-goals. If any of the immediate subgoals does not behave as expected, the user can then issue commands

to examine the subtree descending from that node in more detail. The default command in such circumstances will be to reveal just one more layer down, so that the user can go through the same process at that level, once again expanding any goal nodes which are not behaving as expected, and ignoring any that appear to be correct. In this way, the location of any bug can be found rapidly by means of a top-down, informed depth-first search of the proof tree.

2. Even if the top-level call is to a recursive procedure, the default behaviour is as described in point 1 above, i.e. only the immediate subgoals of the top-level goal would be shown, and any recursive calls to the same procedure below that level would not be made visible unless requested. However, as illustrated by Example 11, it will be possible to issue a command requesting in advance that all calls to a particular procedure or procedures should be made visible.

3. Note that anonymous variables have a numerical suffix added, so that they can be distinguished from one another.

4. To make it easier to see the relative indentation of different goals when the trace output is quite large, the user will be able to request that the space to the left of indented goals (usually blank) should be filled with vertical line characters. This example illustrates the resulting appearance of the trace output.

5. Variable bindings associated with successful evaluations of disjunctive goals are labelled with 'D1' or 'D2', indicating that they resulted from the success of the first or second disjunct, respectively. See for example goals '***6' and '**11'.

```
Program clauses as shown in database window (except for missing 2nd clause
of 'same_path', and any comments, which would not be shown):

1  directed_edge(c,b).
2  directed_edge(c,d).
3  directed_edge(a,b).

1  edge(X, Y):-
    directed_edge(X, Y); directed_edge(Y, X).

1  member(X, [X|_]).
2  member(X, [_|T]):-
    member(X, T).

/*  same_path(X, Y, Forbidden_node_list) is true if X and Y are
```

```
    the same node, or there is a path in the graph from X to Y
    which doesn't contain any of the nodes in
    Forbidden_node_list. */

1  same_path(X, X, _).
   same_path(X, Y, _):- edge(X, Y).  /* As bug, miss out this
                                         clause. */
2  same_path(X, Z, Forbidden_intermediate_nodes):-
    edge(X, Y),
    not(member(Y, Forbidden_intermediate_nodes)),
    same_path(Y, Z, [Y|Forbidden_intermediate_nodes]).

/*  link(X, Y) is true if the nodes X and Y are linked (N.B. any
    node is linked to itself. The 'edge' subgoals here are to
    check that X and Y are somewhere in the graph. 3rd arg. to
    'same_path' is a list of nodes that shouldn't be used in
    finding a path from X to Y - this prevents looping.
*/

1  link(X, Y):-
    edge(X, _),
    edge(Y, _),
    same_path(X, Y, [X,Y]).

Goal: ?- link(a,d).
```

Complete TTT trace, assuming user requests all goals to be shown (65 lines long - the corresponding spy trace is 96 lines):

```
***1: link(a,d)  1Fs
|***2: edge(a, _1)  1SFb
||1    _1 # b
||***3: (directed_edge(a, _1) ; directed_edge(_1, a)) SFb;Fm
|||D1   _1 # b
|||***4: directed_edge(a, _1)  3SFb
||||3    _1 # b
|||**43: directed_edge(_1, a)  Fm
|***5: edge(d, _2)  1SFb
||1    _2 # c
||***6: (directed_edge(d, _2) ; directed_edge(_2, d)) Fm;SFb
|||D2   _2 # c
|||***7: directed_edge(d, _2)  Fm
|||***8: directed_edge(_2, d)  2SFb
```

```
||||2    _2 # c
|***9: same_path(a, d, [a,d])  2Fs
||**10: edge(a, Y_1)  1SFb
|||1    Y_1 # b
|||**11: (directed_edge(a, Y_1) ; directed_edge(Y_1, a)) SFb;Fm
||||D1   Y_1 # b
||||**12: directed_edge(a, Y_1)  3SFb
|||||3    Y_1 # b
||||**42: directed_edge(Y_1, a)  Fm
||**13: not(member(b, [a,d]))  SFb
|||**14: member(b, [a,d])  2Fs
||||**15: member(b, [d])  2Fs
|||||**16: member(b, [])  Fm
||**17: same_path(b, d, [b,a,d])  2Fs
|||**18: edge(b, Y_2)  1SSFb
||||1    Y_2 # c
||||1    Y_2 # a
||||**19: (directed_edge(b, Y_2) ; directed_edge(Y_2, b)) Fm;SSFb
|||||D2   Y_2 # c
|||||D2   Y_2 # a
|||||**20: directed_edge(b, Y_2)  Fm
|||||**21: directed_edge(Y_2, b)  1SFb/3SFb
||||||1    Y_2 # c
||||||3    Y_2 # a
|||**22: not(member(c, [b,a,d]))  SFb
||||**23: member(c, [b,a,d])  2Fs
|||||**24: member(c, [a,d])  2Fs
||||||**25: member(c, [d])  2Fs
|||||||**26: member(c, [])  Fm
|||**27: same_path(c, d, [c,b,a,d])  2Fs
||||**28: edge(c, Y_3)  1SSFb
|||||1    Y_3 # b
|||||1    Y_3 # d
|||||**29: (directed_edge(c, Y_3) ; directed_edge(Y_3, c)) SSFb;Fm
||||||D1   Y_3 # b
||||||D1   Y_3 # d
||||||**30: directed_edge(c, Y_3)  1SFb/2SFb
|||||||1    Y_3 # b
|||||||1    Y_3 # d
||||||**38: directed_edge(Y_3, c)  Fm
||||**31: not(member(b, [c,b,a,d]))  F
|||||**32: member(b, [c,b,a,d])  2S
||||||**33: member(b, [b,a,d])  1S
||||**34: not(member(d, [c,b,a,d]))  F
```

```
|||||**35: member(d, [c,b,a,d])  2S
||||||**36: member(d, [b,a,d])  2S
|||||||**37: member(d, [a,d])  2S
||||||||**38: member(d, [d])  1S
|||**39: not(member(a, [b,a,d]))  F
||||**40: member(a, [b,a,d])  2S
|||||**41: member(a, [a,d])  1S
no
```

```
Development of trace with default restrictions on trace
output (assuming user keeps hitting the ‘return’ key):

***1: link(a, d)  ?

***1: link(a, d)  1?

***1: link(a, d)  1?
 ***2: edge(a, _1)  1S
 |1    _1 = b

***1: link(a, d)  1?
 ***2: edge(a, _1)  1S
 |1    _1 = b
 ***5: edge(d, _2)  ?

***1: link(a, d)  1?
 ***2: edge(a, _1)  1S
 |1    _1 = b
 ***5: edge(d, _2)  1S
 |1    _2 = c
 ***9: same_path(a, d, [a,d])  ?

***1: link(a, d)  1?
 ***2: edge(a, _1)  1S
 |1    _1 = b
 ***5: edge(d, _2)  1S
 |1    _2 = c
 ***9: same_path(a, d, [a,d])  2Fs

At this point, the user may identify the bug, simply by
seeing the result of goal number ‘***9’, and without having
seen any of the distracting details of the subtree involved
in evaluating that goal. However, if the bug is not identified,
```

the user can request more details of the subtree of '***9' to be shown.

```
***1: link(a, d)  1?
 ***2: edge(a, _1)  1S
 |1    _1 = b
 ***5: edge(d, _2)  1S?
 |1    _2 # c
 ***9: same_path(a, d, [a,d])  2Fs

***1: link(a, d)  1?
 ***2: edge(a, _1)  1S
 |1    _1 = b
 ***5: edge(d, _2)  1SFb
 |1    _2 # c
 ***9: same_path(a, d, [a,d])  2Fs

***1: link(a, d)  1?
 ***2: edge(a, _1)  1S?
 |1    _1 # b
 ***5: edge(d, _2)  1SFb
 |1    _2 # c
 ***9: same_path(a, d, [a,d])  2Fs

***1: link(a, d)  1?
 ***2: edge(a, _1)  1SFb
 |1    _1 # b
 ***5: edge(d, _2)  1SFb
 |1    _2 # c
 ***9: same_path(a, d, [a,d])  2Fs

***1: link(a, d)  1Fs
 ***2: edge(a, _1)  1SFb
 |1    _1 # b
 ***5: edge(d, _2)  1SFb
 |1    _2 # c
 ***9: same_path(a, d, [a,d])  2Fs
```

```
***1: link(a, d)  1Fs
 ***2: edge(a, _1)  1SFb
 |1    _1 # b
 ***5: edge(d, _2)  1SFb
 |1    _2 # c
 ***9: same_path(a, d, [a,d])  2Fs
no
```

Example 11: Focussing attention on a predicate.

In the spy and EPTB tracers, it is possible to make visible only calls to certain specified procedures. A similar facility will be available in the TTT, although in this case, it is suggestedthat the general principle of minimising the amount of trace output should be waived slightly, so that not just the calls to the specified procedures are shown, but also the siblings and immediate subgoals of any such calls, together with the top-level goal and its immediate subgoals. The effect of this approach is that calls to the specified procedure or procedures are not seen in total isolation, but rather as they are located within the overall context of the computation, which is arguably useful for debugging purposes in many instances. The resulting trace is still typically quite small, being usually considerably smaller than a trace showing all calls.

It will be possible to alter dynamically (i.e. during the execution of the trace) the procedure or procedures focussed upon — a feature not available in either the spy, the EPTB or the TPM.

This example illustrates the 'focussing' process in concrete terms. It is assumed that the user has requested that every call to the procedure 'same_path' should be displayed: the tracer will then output the minimum 'context tree' necessary to show all such calls . The final length of the trace is 23 lines, as opposed to a length of 65 lines for the full trace (which was shown previously at the start of Example 10).

```
Program clauses and goal as in Example 10.

Complete TTT trace:

***1: link(a, d)  1Fs
 ***2: edge(a, _1)  1SFb
 |1    _1 # b
 ***5: edge(d, _2)  1SFb
 |1    _2 # c
 ***9: same_path(a, d, [a,d])  2Fs
  **10: edge(a, Y_1)  1SFb
```

```
  |1    Y_1 # b
  **13: not(member(b, [a,d]))  SFb
  **17: same_path(b, d, [b,a,d])  2Fs
   **18: edge(b, Y_2)  1SSFb
   |1    Y_2 # c
   |1    Y_2 # a
   **22: not(member(c, [b,a,d]))  SFb
   **27: same_path(c, d, [c,b,a,d])  2Fs
    **28: edge(c, Y_3)  1SSFb
    |1    Y_3 # b
    |1    Y_3 # d
    **31: not(member(b, [c,b,a,d]))  F
    **34: not(member(d, [c,b,a,d]))  F
   **39: not(member(a, [b,a,d]))  F
no
```

4.6 Display of extra information

As described earlier in section 4.1.5, the user will be able to request — either prospectively or retrospectively — the display of additional information beyond the basic, default output. The examples in this section indicate some possible kinds of additional information that could be requested.

Example 12: Showing immediate subgoals of a goal.

```
Program clauses and goal: as Example 10.

Trace output before command:

***1: link(a, d)  1?
 ***2: edge(a, _1)  1S     <---- goal selected
 |1    _1 = b
 ***5: edge(d, _2)  1S
 |1    _2 = c

Trace output after command:

***1: link(a, d)  1?
 ***2: edge(a, _1)  1S     <---- goal selected
 |1    _1 = b
  ***3:(directed_edge(a, _1) ; directed_edge(_1, a))  S;
 ***5: edge(d, _2)  1S
 |1    _2 = c
```

Example 13: Showing entire subtree of a goal.

```
Program clauses and goal: as Example 10.

Trace output before command:

***1: link(a,d)  1?
 ***2: edge(a, _1)  1S     <---- goal selected
 |1    _1 = b
 ***5: edge(d, _2)  1S
 |1    _2 = c
 ***9: same_path(a, d, [a,d])  ?

Trace output after command:

***1: link(a,d)  1?
 ***2: edge(a, _1)  1S
 |1    _1 = b
  ***3: (directed_edge(a, _1) ; directed_edge(_1, a))  S;
  |D1   _1 = b
   ***4: directed_edge(a, _1)  3S
   |3    _1 = b
 ***5: edge(d, _2)  1S
 |1    _2 = c
 ***9: same_path(a, d, [a,d])  ?
```

Example 14: Showing instantiation of a clause.

As illustrated by this example, the TTT will use the same method as the EPTB for showing the instantiation of a clause and the bindings of its variables. Points to note:

1. The clause before and after instantiation resulting from matching with a goal is represented by two adjacent lines with '....' prefixes, so that they stand out clearly from the lines representing goals, which have prefixes containing '*' characters, such as '***2'.

2. Clause variables are shown as they appear in the clauses of the program, except for the addition of numerical suffixes. These suffixes allow clause variables associated with different invocations of the same clause to be clearly distinguishable from one another.

3. The bindings of clause variables after matching with a goal can be seen by comparing the corresponding arguments in the lines representing the

clause before and after matching. To facilitate comparison, the corresponding arguments are shown in vertical alignment with one another. For example, immediately after the head of clause 2 of 'conc' matches goal '***2', the variable H_2 is bound to the value 'b'; the variable L_2 is bound to [c]; and the variable T1_2 is unbound, since it is shown simply as 'T1_2' in both 'before matching' and 'after matching' lines.

4. Each pair of lines showing the instantiation of a clause before and after matching with a goal is immediately preceded by a line containing a 'clause status field' (e.g. the line '|1?' in this example). This field serves to identify the clause involved, and is identical to the corresponding subfield of the associated goal's 'goal status field'. Thus in this example, '1?' appears also in the goal status field of the goal numbered '***2'. The clause status field has a line to itself, because if a clause has been resatisfied several times on backtracking, the sequence of 'S' characters indicating this may require more space than would be available if the clause status field were shown as a prefix on one of the two lines displaying the clause before or after matching with a goal.

```
Program as shown in 'database window':

1  conc([H|T], L, [H|T1]):-
    conc(T, L, T1).
2  conc([], L, L).

Goal: ?- conc([a,b], [c], L).

TTT trace before command:

***1: conc([a,b], [c], L)  1?
 ***2: conc([b], [c], T1_1)  1?   <-- Clause 2 of this goal selected
  ***3: conc([], [c], T1_2)  2S        by cursor on '2'
  |2    T1_2 = [c]

TTT trace after command:

***1: conc([a,b], [c], L)  1?
 ***2: conc([b], [c], T1_1)  1?
 |1?
 |.... conc([H_2|T_2], L_2, [H_2|T1_2]):- conc(T_2, L_2, T1_2).
 |.... conc([b  |[] ], [c], [b  |T1_2]):- conc([] , [c], T1_2).
  ***3: conc([], [c], T1_2)  2S
  |2    T1_2 = [c]
```

Example 15: Showing current binding history of a variable.

Points to note:

1. On request from the user, successive bindings of a variable in the course of a single attempt to satisfy a goal are shown separated by '=' characters (as for example the two bindings of 'L' in this example). Usually a single line will suffice to show the sequence of bindings, but if not, additional lines may be added as necessary. The ordering of binding values is such that the rightmost value in the lowermost line of values for a variable represents its most recent binding (e.g. in this trace, '[a,b|T1_2]' is the most recent binding of the variable 'L').

```
Program and goal as Example 14.

TTT trace before command:

***1: conc([a,b], [c], L)  1?   <--- Variable L selected by cursor
 ***2: conc([b], [c], T1_1)  1?        on 'L'.
  ***3: conc([], [c], T1_2)  2S
  |2    T1_2 = [c]

TTT trace after command:

***1: conc([a,b], [c], L)  1?
|2    L = [a|T1_1] = [a,b|T1_2]
 ***2: conc([b], [c], T1_1)  1?
  ***3: conc([], [c], T1_2)  2S
  |2    T1_2 = [c]
```

Example 16: Showing full details as trace advances.

```
Program and goal as Example 14.

Development of the TTT trace:

***1: conc([a,b], [c], L)  ?

***1: conc([a,b], [c], L)  1?
|1    L = [a|T1_1] =
|1?
|.... conc([H_1|T_1], L_1, [H_1|T1_1]):- conc(T_1, L_1, T1_1).
|.... conc([a  |[b]], [c], [a  |T1_1]):- conc([b], [c], T1_1).
```

```
***1: conc([a,b], [c], L)  1?
|1    L = [a|T1_1] =
|1?
|.... conc([H_1|T_1], L_1, [H_1|T1_1]):- conc(T_1, L_1, T1_1).
|.... conc([a  |[b]], [c], [a  |T1_1]):- conc([b], [c], T1_1).
 ***2: conc([b], [c], L)  ?

***1: conc([a,b], [c], L)  1?
|1    L = [a|T1_1] =
|1?
|.... conc([H_1|T_1], L_1, [H_1|T1_1]):- conc(T_1, L_1, T1_1).
|.... conc([a  |[b]], [c], [a  |T1_1]):- conc([b], [c], T1_1).
 ***2: conc([b], [c], T1_1)  1?
 |1    T1_1 = [b|T1_2] =
 |1?
 |.... conc([H_2|T_2], L_2, [H_2|T1_2]):- conc(T_2, L_2, T1_2).
 |.... conc([b  |[] ], [c], [b  |T1_2]):- conc([],  [c], T1_2).

***1: conc([a,b], [c], L)  1?
|1    L = [a|T1_1] = [a,b|T1_2] =
|1?
|.... conc([H_1|T_1], L_1, [H_1|T1_1]):- conc(T_1, L_1, T1_1).
|.... conc([a  |[b]], [c], [a  |T1_1]):- conc([b], [c], T1_1).
 ***2: conc([b], [c], T1_1)  1?
 |1    T1_1 = [b|T1_2] =
 |1?
 |.... conc([H_2|T_2], L_2, [H_2|T1_2]):- conc(T_2, L_2, T1_2).
 |.... conc([b  |[] ], [c], [b  |T1_2]):- conc([],  [c], T1_2).

***1: conc([a,b], [c], L)  1?
|1    L = [a|T1_1] = [a,b|T1_2] =
|1?
|.... conc([H_1|T_1], L_1, [H_1|T1_1]):- conc(T_1, L_1, T1_1).
|.... conc([a  |[b]], [c], [a  |T1_1]):- conc([b], [c], T1_1).
 ***2: conc([b], [c], T1_1)  1?
 |1    T1_1 = [b|T1_2] =
 |1?
 |.... conc([H_2|T_2], L_2, [H_2|T1_2]):- conc(T_2, L_2, T1_2).
 |.... conc([b  |[] ], [c], [b  |T1_2]):- conc([],  [c], T1_2).
  ***3: conc([], [c], T1_2)  ?
```

```
***1: conc([a,b], [c], L)  1?
|1    L = [a|T1_1] = [a,b|T1_2] =
|1?
|.... conc([H_1|T_1], L_1, [H_1|T1_1]):- conc(T_1, L_1, T1_1).
|.... conc([a  |[b]], [c], [a  |T1_1]):- conc([b], [c], T1_1).
 ***2: conc([b], [c], T1_1)  1?
 |1    T1_1 = [b|T1_2] =
 |1?
 |.... conc([H_2|T_2], L_2, [H_2|T1_2]):- conc(T_2, L_2, T1_2).
 |.... conc([b  |[] ], [c], [b  |T1_2]):- conc([],  [c], T1_2).
  ***3: conc([], [c], T1_2)  2?
  |2    T1_2 = [c] =
  |2?
  |.... conc([], L_3, L_3).
  |.... conc([], [c], [c]).

***1: conc([a,b], [c], L)  1?
|1    L = [a|T1_1] = [a,b|T1_2] =
|1?
|.... conc([H_1|T_1], L_1, [H_1|T1_1]):- conc(T_1, L_1, T1_1).
|.... conc([a  |[b]], [c], [a  |T1_1]):- conc([b], [c], T1_1).
 ***2: conc([b], [c], T1_1)  1?
 |1    T1_1 = [b|T1_2] = [b,c] =
 |1?
 |.... conc([H_2|T_2], L_2, [H_2|T1_2]):- conc(T_2, L_2, T1_2).
 |.... conc([b  |[] ], [c], [b  |T1_2]):- conc([],  [c], T1_2).
  ***3: conc([], [c], T1_2)  2?
  |2    T1_2 = [c] =
  |2?
  |.... conc([], L_3, L_3).
  |.... conc([], [c], [c]).
```

```
***1: conc([a,b], [c], L)  1?
|1    L = [a|T1_1] = [a,b|T1_2] = [a,b,c] =
|1?
|.... conc([H_1|T_1], L_1, [H_1|T1_1]):- conc(T_1, L_1, T1_1).
|.... conc([a|[b]],   [c], [a  |T1_1]):- conc([b], [c], T1_1).
 ***2: conc([b], [c], T1_1)  1?
 |1    T1_1 = [b|T1_2] = [b,c] =
 |1?
 |.... conc([H_2|T_2], L_2, [H_2|T1_2]):- conc(T_2, L_2, T1_2).
 |.... conc([b  |[] ], [c], [b  |T1_2]):- conc([],  [c], T1_2).
  ***3: conc([], [c], T1_2)  2?
  |2    T1_2 = [c] =
  |2?
  |.... conc([], L_3, L_3).
  |.... conc([], [c], [c]).

***1: conc([a,b], [c], L)  1?
|1    L = [a|T1_1] = [a,b|T1_2] = [a,b,c] =
|1?
|.... conc([H_1|T_1], L_1, [H_1|T1_1]):- conc(T_1, L_1, T1_1).
|.... conc([a  |[b]], [c], [a  |T1_1]):- conc([b], [c], T1_1).
 ***2: conc([b], [c], T1_1)  1?
 |1    T1_1 = [b|T1_2] = [b,c] =
 |1?
 |.... conc([H_2|T_2], L_2, [H_2|T1_2]):- conc(T_2, L_2, T1_2).
 |.... conc([b  |[] ], [c], [b  |T1_2]):- conc([],  [c], T1_2).
  ***3: conc([], [c], T1_2)  2S
  |2    T1_2 = [c]
  |2S
  |.... conc([], L_3, L_3).
  |.... conc([], [c], [c]).
```

```
***1: conc([a,b], [c], L)  1?
|1    L = [a|T1_1] = [a,b|T1_2] = [a,b,c] =
|1?
|.... conc([H_1|T_1], L_1, [H_1|T1_1]):- conc(T_1, L_1, T1_1).
|.... conc([a  |[b]], [c], [a  |T1_1]):- conc([b], [c], T1_1).
 ***2: conc([b], [c], T1_1)  1S
 |1    T1_1 = [b|T1_2] = [b,c]
 |1S
 |.... conc([H_2|T_2], L_2, [H_2|T1_2]):- conc(T_2, L_2, T1_2).
 |.... conc([b  |[] ], [c], [b  |T1_2]):- conc([],  [c], T1_2).
  ***3: conc([], [c], T1_2)  2S
  |2    T1_2 = [c]
  |2S
  |.... conc([], L_3, L_3).
  |.... conc([], [c], [c]).

***1: conc([a,b], [c], L)  1S
|1    L = [a|T1_1] = [a,b|T1_2] = [a,b,c]
|1S
|.... conc([H_1|T_1], L_1, [H_1|T1_1]):- conc(T_1, L_1, T1_1).
|.... conc([a  |[b]], [c], [a  |T1_1]):- conc([b], [c], T1_1).
 ***2: conc([b], [c], T1_1)  1S
 |1    T1_1 = [b|T1_2] = [b,c]
 |1S
 |.... conc([H_2|T_2], L_2, [H_2|T1_2]):- conc(T_2, L_2, T1_2).
 |.... conc([b  |[] ], [c], [b  |T1_2]):- conc([],  [c], T1_2).
  ***3: conc([], [c], T1_2)  2S
  |2    T1_2 = [c]
  |2S
  |.... conc([], L_3, L_3).
  |.... conc([], [c], [c]).
```

```
***1: conc([a,b], [c], L)  1S
|1    L = [a|T1_1] = [a,b|T1_2] = [a,b,c]
|1S
|.... conc([H_1|T_1], L_1, [H_1|T1_1]):- conc(T_1, L_1, T1_1).
|.... conc([a|[b]],   [c], [a  |T1_1]):- conc([b], [c], T1_1).
 ***2: conc([b], [c], T1_1)  1S
 |1    T1_1 = [b|T1_2] = [b,c]
 |1S
 |.... conc([H_2|T_2], L_2, [H_2|T1_2]):- conc(T_2, L_2, T1_2).
 |.... conc([b  |[] ], [c], [b  |T1_2]):- conc([],  [c], T1_2).
  ***3: conc([], [c], T1_2)  2S
  |2    T1_2 = [c]
  |2S
  |.... conc([], L_3, L_3).
  |.... conc([], [c], [c]).
yes
```

4.7 Handling of assertion and retraction

One of the features available in the TTT will be its use of the 'database window' to show dynamic changes to the database brought about by the assertion and retraction of clauses. (N.B. The Prolog database can also be changed as a result of using the system predicate 'consult' — the tracer would have to be able to deal with that as well, although the use of 'consult' is not considered in this report.) This will be particularly useful for dealing with faulty programs involving database modification, which are often difficult to debug with some other tracers.

The method used for labelling the clauses of a procedure, illustrated by Examples 17 to 19, allows the original and current clause sets for a procedure to be easily identifiable. Each clause of a procedure is assigned a unique identifying number, which it retains throughout the execution of the program, so that different invocations of the same clause are readily identified as involving the same clause, even from the trace window alone. (This is important, because it means that a hard copy of the trace of the program can supply this information on its own, without the need for any hard copies of the database window.)

Example 17: Assertion using 'assert' or 'assertz'.

Points to note:

1. If a clause has been added to the database using 'assert' or 'assertz' — both of which will add it to the bottom of the database — this is indicated by a "z" next to its identifying number.

```
Initial p clauses as shown in database window:

1    p(a, _).
2    p(b, c).
3    p(d, c).
4    p(d, d).
5    p(e, f).

Goal: ?- p(b, X), assert(p(e,g)), p(Y,g).
(N.B. 'assertz' would have the same effect here.)

Development of the TTT trace and database windows:

(N.B. The two windows will not appear side by side in the
tracer - if Ved is being used, one window will be above the
other, and on a Sun workstation, the windows could be
Sunview windows)

 Trace window                         | Database window
                                      |
***1: assert(p(e,g)), p(Y,g)  ?       | 1    p(a, _).
                                      | 2    p(b, c).
                                      | 3    p(d, c).
                                      | 4    p(d, d).
                                      | 5    p(e, f).
                                      |
***1: assert(p(e,g)), p(Y,g)  ?       | 1    p(a, _).
 ***2: assert(p(e,g))  ?              | 2    p(b, c).
                                      | 3    p(d, c).
                                      | 4    p(d, d).
                                      | 5    p(e, f).
                                      |
***1: assert(p(e,g)), p(Y,g)  S,?     | 1    p(a, _).
 ***2: assert(p(e,g))  S              | 2    p(b, c).
                                      | 3    p(d, c).
                                      | 4    p(d, d).
                                      | 5    p(e, f).
                                      | 6z   p(e, g).
```

```
***1: assert(p(e,g)), p(Y,g)  S,?    | 1    p(a, _).
 ***2: assert(p(e,g))  S             | 2    p(b, c).
 ***3: p(Y,g)  ?                     | 3    p(d, c).
                                     | 4    p(d, d).
                                     | 5    p(e, f).
                                     | 6z   p(e, g).
                                     |
***1: assert(p(e,g)), p(Y,g)  S,?    | 1    p(a, _).
 ***2: assert(p(e,g))  S             | 2    p(b, c).
 ***3: p(Y,g)  6z?                   | 3    p(d, c).
                                     | 4    p(d, d).
                                     | 5    p(e, f).
                                     | 6z   p(e, g).
                                     |
***1: assert(p(e,g)), p(Y,g)  S,6zS  | 1    p(a, _).
 ***2: assert(p(e,g))  S             | 2    p(b, c).
 ***3: p(Y,g)  6zS                   | 3    p(d, c).
 |6z   Y = e                         | 4    p(d, d).
                                     | 5    p(e, f).
                                     | 6z   p(e, g).
                                     |
***1: assert(p(e,g)), p(Y,g)  S,6zS  | 1    p(a, _).
 ***2: assert(p(e,g))  S             | 2    p(b, c).
 ***3: p(Y,g)  6zS                   | 3    p(d, c).
 |6z   Y = e                         | 4    p(d, d).
yes                                  | 5    p(e, f).
                                     | 6z   p(e, g).
                                     |
```

Example 18: Assertion using 'asserta'.

Points to note:

1. Regardless of whether it is asserted at the start or end of the database, the first asserted clause of a procedure is given an identifying number one greater than the number of the last of the original clauses for that procedure. Thus in this example, in which there are 5 clauses for 'p' in the database initially, the fresh clause is given the number 6.

2. If a clause has been asserted using 'asserta', this is indicated in the database and trace windows by an 'a' character next to its identifying number (as in '6a' in this example).

3. Following the assertion or retraction of a clause, there is no renumbering of any of the other clauses for the same procedure. However, if the ordinal

position of any other clause is altered as a result, and such a clause is tried subsequently, the trace window indicates its ordinal position in the database at the time of calling in parentheses, after its unique identifying number. Hence in this example, after the assertion of clause '6a' of 'p', clause 2 becomes the third clause, and so its second invocation is indicated by the appearance of '2(3)' in the trace output.

4. Note that the original set of clauses for a procedure can always be easily identified in the 'database window': its members are the clauses whose identifying numbers have no 'a' or 'z' suffix attached.

```
Program: as Example 17.

Goal: ?- p(b,c), asserta(p(e,g)), p(b,c).

Development of the TTT trace and database windows:

 Trace window                                     | Database window
                                                  |
***1: p(b,c), asserta(p(e,g)), p(b,c)  ?          | 1    p(a, _).
                                                  | 2    p(b, c).
                                                  | 3    p(d, c).
                                                  | 4    p(d, d).
                                                  | 5    p(e, f).
                                                  |
                                                  |
***1: p(b,c), asserta(p(e,g)), p(b,c)  ?          | 1    p(a, _).
 ***2: p(b,c)  ?                                  | 2    p(b, c).
                                                  | 3    p(d, c).
                                                  | 4    p(d, d).
                                                  | 5    p(e, f).
                                                  |
                                                  |
***1: p(b,c), asserta(p(e,g)), p(b,c)  2?         | 1    p(a, _).
 ***2: p(b,c)  2?                                 | 2    p(b, c).
                                                  | 3    p(d, c).
                                                  | 4    p(d, d).
                                                  | 5    p(e, f).
```

```
***1: p(b,c), asserta(p(e,g)), p(b,c)  2S,?       | 1    p(a, _).
 ***2: p(b,c)  2S                                 | 2    p(b, c).
                                                  | 3    p(d, c).
                                                  | 4    p(d, d).
                                                  | 5    p(e, f).
                                                  |
                                                  |
***1: p(b,c), asserta(p(e,g)), p(b,c)  2S,?       | 1    p(a, _).
 ***2: p(b,c)  2S                                 | 2    p(b, c).
 ***3: asserta(p(e,g))  ?                         | 3    p(d, c).
                                                  | 4    p(d, d).
                                                  | 5    p(e, f).
                                                  |
                                                  |
***1: p(b,c), asserta(p(e,g)), p(b,c)  2S,S,?     | 6a   p(e, g).
 ***2: p(b,c)  2S                                 | 1    p(a, _).
 ***3: asserta(p(e,g))  S                         | 2    p(b, c).
                                                  | 3    p(d, c).
                                                  | 4    p(d, d).
                                                  | 5    p(e, f).
                                                  |
                                                  |
***1: p(b,c), asserta(p(e,g)), p(b,c)  2S,S,?     | 6a   p(e, g).
 ***2: p(b,c)  2S                                 | 1    p(a, _).
 ***3: asserta(p(e,g))  S                         | 2    p(b, c).
 ***4: p(b,c)  ?                                  | 3    p(d, c).
                                                  | 4    p(d, d).
                                                  | 5    p(e, f).
                                                  |
                                                  |
***1: p(b,c), asserta(p(e,g)), p(b,c)  2S,S,2(3)? | 6a   p(e, g).
 ***2: p(b,c)  2S                                 | 1    p(a, _).
 ***3: asserta(p(e,g))  S                         | 2    p(b, c).
 ***4: p(b,c)  2(3)?                              | 3    p(d, c).
                                                  | 4    p(d, d).
                                                  | 5    p(e, f).
                                                  |
                                                  |
***1: p(b,c), asserta(p(e,g)), p(b,c)  2S,S,2(3)S | 6a   p(e, g).
 ***2: p(b,c)  2S                                 | 1    p(a, _).
 ***3: asserta(p(e,g))  S                         | 2    p(b, c).
 ***4: p(b,c)  2(3)S                              | 3    p(d, c).
                                                  | 4    p(d, d).
                                                  | 5    p(e, f).
```

```
***1: p(b,c), asserta(p(e,g)), p(b,c)  2S,S,2(3)S | 6a   p(e, g).
 ***2: p(b,c)  2S                                 | 1    p(a, _).
 ***3: asserta(p(e,g))  S                         | 2    p(b, c).
 ***4: p(b,c)  2(3)S                              | 3    p(d, c).
yes                                               | 4    p(d, d).
                                                  | 5    p(e, f).
```

Example 19: Retraction.

Points to note:

1. When a clause is retracted, it is still shown in the database window, but is marked with a # character to show that it is no longer current. Hence the current set of clauses is easily identifiable from the 'database window': its members are the clauses with no # prefix to their identifying number.

2. The clause heads are aligned with each other, and the clause numbers are left-justified in a field wide enough to accommodate clause labels such as '999z'. To maintain readability, it is important to have at least one blank column preceding the clause heads: hence using a # suffix to the clause label rather than a # prefix would not save any space, because it would require a two-column gap between clause labels and clause heads. Also, a # character at the start of a line stands out more clearly than one somewhere in the middle of a line.

```
Program: as Example 17.

Goal: ?- retract(p(X,c)), p(d,d).

Development of the TTT trace and database windows:

  Trace window                              | Database window
                                            |
***1: retract(p(X,c)), p(d,d)  ?            | 1    p(a, _).
                                            | 2    p(b, c).
                                            | 3    p(d, c).
                                            | 4    p(d, d).
                                            | 5    p(e, f).
```

```
***1: retract(p(X,c)), p(d,d)  ?            | 1    p(a, _).
 ***2: retract(p(X,c))  ?                   | 2    p(b, c).
                                            | 3    p(d, c).
                                            | 4    p(d, d).
                                            | 5    p(e, f).
                                            |
                                            |
***1: retract(p(X,c)), p(d,d)  S,?          | 1    p(a, _).
 ***2: retract(p(X,c))  S                   |#2    p(b, c).
 |     X = b                                | 3    p(d, c).
                                            | 4    p(d, d).
                                            | 5    p(e, f).
                                            |
                                            |
***1: retract(p(X,c)), p(d,d)  S,?          | 1    p(a, _).
 ***2: retract(p(X,c))  S                   |#2    p(b, c).
 |     X = b                                | 3    p(d, c).
 ***3: p(d,d)  ?                            | 4    p(d, d).
                                            | 5    p(e, f).
                                            |
                                            |
***1: retract(p(X,c)), p(d,d)  S,4(3)?      | 1    p(a, _).
 ***2: retract(p(X,c))  S                   |#2    p(b, c).
 |     X = b                                | 3    p(d, c).
 ***3: p(d,d)  4(3)?                        | 4    p(d, d).
                                            | 5    p(e, f).
                                            |
                                            |
***1: retract(p(X,c)), p(d,d)  S,4(3)S      | 1   p(a, _).
 ***2: retract(p(X,c))  S                   |#2   p(b, c).
 |     X = b                                | 3   p(d, c).
 ***3: p(d,d)  4(3)S                        | 4   p(d, d).
                                            | 5   p(e, f).
                                            |
                                            |
***1: retract(p(X,c)), p(d,d)  S,4(3)S      | 1   p(a, _).
 ***2: retract(p(X,c))  S                   |#2   p(b, c).
 |     X = b                                | 3   p(d, c).
 ***3: p(d,d)  4(3)S                        | 4   p(d, d).
yes                                         | 5   p(e, f).
```

5 Summary

Three existing tracers — the standard 'spy' tracer, the experimental EPTB tracer and the CDL-TPM, a commercially available version of the graphical TPM tracer — have been compared and criticised according to their general features and characteristics, and their adherence to a set of informal design principles. Evaluation of these tracers has led to the conception of a new tracer, which incorporates the best aspects from them, together with some additional features, whilst avoiding some of their shortcomings.

The new tracer, known as the 'Textual Tree Tracer' (TTT), will be textual, non-linear, and indented. It will use a 'sideways tree' representation of the proof tree, which facilitates correlation of the trace with the program clauses, as displayed in a separate 'database window'. Like the EPTB tracer, the TTT will make available a considerable amount of information concerning variable bindings and the matching of clauses — although only the most essential information will be shown automatically. Like the TPM, the TTT will show very clearly the flow of control, including backtracking and the effect of the cut, by its use of a tree representation. A characteristic feature of the TTT will be its extensive use of default controls to restrict the quantity of trace output made visible. When combined with a basic notation that is already very compact — producing in many cases a trace shorter than the 'spy' trace, even when all goals are shown — this feature of default restriction will lead to very concise program traces, which will help to focus the attention of the user by minimising the amount of distracting information, thereby making it easier to locate bugs more quickly.

References

[Byrd 80] Byrd, L. (1980). Understanding the control flow of Prolog programs. In S-A Tarnlund (Ed.)., Proceedings of the Logic Programming Workshop, Debrecen, Hungary.

[Clocksin & Mellish 81] Clocksin, W.F. & Mellish, C.S. (1981). Programming in Prolog. Springer-Verlag.

[Dewar & Cleary 86] Dewar, A.D. & Cleary, J.G. (1986). Graphical display of complex information within a Prolog debugger. International Journal of Man Machine Studies, 25, 503-521, 1986.

[Dichev & du Boulay 89] Dichev, C. & du Boulay, B. (1989). An enhanced trace tool for Prolog. Cognitive Science Research Paper No.138, University of Sussex.

[Eisenstadt 84] Eisenstadt, M. (1984). A powerful Prolog trace package. Proceedings of the Sixth European Conference on Artificial Intelligence (ECAI-84), Pisa, Italy, 1984.

[Eisenstadt, Hasemer & Kriwaczek] Eisenstadt, M., Hasemer, T., & Kriwaczek, F. (1985). An improved user-interface for Prolog. In B. Shackel (Ed.), Human-Computer Interaction. Amsterdam: Elsevier (North-Holland), 1985.

[Eisenstadt & Brayshaw 88] Eisenstadt, M. & Brayshaw, M. (1988). The transparent Prolog machine (TPM): An execution model and graphical debugger for logic programming. Journal of Logic Programming, 5 (4), 1988, pp. 277-342.

[Rajan 86] Rajan, T. (1986). The design of animated tracing tools for novice programmers. Technical Report 16, Human Cognition Research Laboratory, The Open University, Milton Keynes.

[Rajan 90] Rajan, T. (1990). Principles for the design of dynamic tracing environments for novice programmers. Instructional Science, 19 (4/5), pp. 377-406, 1990.