

/PEDIT -
A Resident Structure Editor
for PROLOG/

by

SANDRA LEE DUFFY

B.S. University of Illinois, 1977

A MASTER'S REPORT

submitted in partial fulfillment of the

requirements for the degree

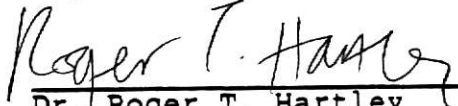
MASTER OF SCIENCE

Department of Computer Science

KANSAS STATE UNIVERSITY
Manhattan, Kansas

1985

Approved by:


Dr. Roger T. Hartley

LD
2668
IR4
1985
D83
C.2

PEDIT -
A Resident Structure Editor
for PROLOG

A11202 996282

CONTENTS

1. Chapter One.....	1
1.1 Introduction.....	1
1.2 Literature Review.....	3
2. Chapter Two.....	12
2.1 Requirements.....	12
3. Chapter Three.....	16
3.1 Design.....	16
4. Chapter Four.....	27
4.1 Implementation.....	27
4.2 Testing.....	32
5. Chapter Five.....	33
5.1 Conclusions.....	33
5.2 Extensions.....	34
BIBLIOGRAPHY.....	37
Glossary.....	39
PEDIT Commands.....	40
USER'S MANUAL.....	42
Source Code.....	90

PEDIT -
A Resident Structure Editor
for PROLOG

LIST OF FIGURES

Figure 1.	Program Flow.....	17
Figure 2.	Structure Tree.....	18
Figure 3.	Rule Tree.....	19
Figure 4.	Fact Tree.....	19
Figure 5.	PROLOG Structure.....	20
Figure 6.	Rule Structure.....	20
Figure 7.	Internal PEDIT Format.....	20
Figure 8.	Goal Tree.....	22
Figure 9.	Input Command Flow.....	29

1. Chapter One

PROLOG was created around 1970 [5] and as such is a relatively new programming language. This compares to 1962 for the creation of LISP, a language with similar applications. Due to its "youth", some of the utilities taken for granted in most programming environments are not available in the PROLOG programming environment. For example, an editor based within the PROLOG environment is non-existent to this author's knowledge. This paper describes PEDIT - a resident structure editor for PROLOG. Familiarity with PROLOG use and format by the reader is assumed. The implementation of PEDIT is intended to fill the editor gap in the PROLOG programming environment.

1.1 Introduction

Currently, a programmer debugging a program within the PROLOG environment has two options for making program changes. First, the programmer might exit the interpreter and edit the program file(s) using a text editor such as "vi" or "ed". Then, the interpreter must be reentered and the program file(s) reloaded. This iteration usually occurs many times before a program is correct. This method is both time-consuming and costly on any size program. On a large program, the costs could be prohibitive.

It should be noted that PROLOG commands can be created to edit program files without exiting the interpreter. This would give the impression of an interpreter-based editor; however, this method merely simulates the method mentioned above. It would not avoid the cost of reloading the edited files into the PROLOG environment.

A second method available to the PROLOG programmer is the manual deletion and addition of entire PROLOG clauses.* This could be accomplished with the use of the "retract" or "abolish" commands together with the "assert" commands. This method, however, would probably require the deletion and insertion of entire procedures due to the manner the "assert" commands operate.

If a single clause were "retract"ed and then "assert"ed, it would lose its position in the data base in relation to the other clauses for that procedure. Because PROLOG performs a "top-down" traversal of the data base, the logic for that procedure would in all probability be severely modified. Thus, entire procedures would need to be deleted and then inserted to ensure proper ordering of clauses. As can be seen, this method would only be practical if the entire

* NOTE: Appendix A contains a glossary of PROLOG terminology used in this paper.

procedure were faulty. If only a small error existed in a procedure, this method would be too cumbersome and time-consuming. Also, to an inexperienced typist, it could indeed resemble Purgatory.

Due to the reasons listed above, plus the enticement of enhancing an "infant" language, the task was undertaken to write a PROLOG-based editor. Since PROLOG is replacing LISP in some areas and thus is growing in use, it was agreed that this task would indeed fill a need of the future.

Once the task was undertaken, a decision was needed on what type of editor was appropriate. Since no PROLOG editor was known to exist, it was not possible to base the editor on other PROLOG editors. First, a general comparison of editor types was conducted. Second, since PROLOG and LISP are both used for Artificial Intelligence applications and as such are often used by the same personnel, thought was given to basing the PROLOG editor on existing LISP editors. Thus, a comparison of LISP editors was also conducted.

1.2 Literature Review

The merits and deficiencies of different types of editors have been debated since their invention. The viewpoints of various authors are presented and summarized here.

1.2.1 Text vs. Structure Editors For the purposes of this paper, a text editor is here defined as an editor which performs only text editing functions. A structure editor is defined as having the capability of parsing and modifying tree structures. In other words, a structure can be traversed by moving a cursor around. A pure structure editor does not include text editing capabilities. Instead, portions of trees are deleted and replaced using tools such as templates. A hybrid editor incorporates both text editing and structure editing commands.

Structure editors are considered by most to be far superior to text editors. There are, of course, others who claim that structure editors do not entirely fill the editing needs of a programmer.

Among those who praise structure editors is Wilander [20]. He says that the advantage of structure editors is that only the parts affected by an edit are changed. This means that the structure of the program remains intact since a structure editor only allows logical portions of a program to be changed.

In his paper, Waters [19] contends that structure editors do not satisfy all the needs of an editor. He proposes that text-oriented commands not be abandoned in a structure editor environment. He argues for a hybrid approach to

editing in order to incorporate the advantages of both the textual and structural viewpoints. To support his point, Waters points to EMACS [15] which has supported both text and structure-oriented commands for more than a decade. Waters points out that although many claim that new methods are superior to old methods, this very seldom is the case. He says that new methods are liable to leave unaddressed some of the issues handled by the old methods.

Teitelbaum and Reps [17] support the basic premise made by Waters. The Cornell Program Synthesizer is a hybrid design which they claim seeks a pragmatic balance between the extremes of a structure editor and a text editor. They state that although they promote the structural perspective, they recognize that text-oriented commands are sometimes needed. They do mention, however, that at times contradictions arise between the two perspectives which can lead to confusion and inconvenience. They list examples of this and conclude that mechanisms can be incorporated into editors such as the Synthesizer to minimize the contradictions by enforcing user actions.

Stromfors and Jonesjo [16] describe ED3, a hybrid editor which they call a structure-oriented text editor. They describe their method of superimposing a tree structure onto text which is then applicable to any type of object, whether it is text, figures or data records. The structure editing

commands parse the tree. When the desired tree position is reached, the text editor commands can be utilized. Stromfors and Jonesjo provide scenarios for the application of the ED3 editor and mention that it is especially useful for handling the source text of large program systems and for the preparation of structured documents.

Shani [13] refutes the points supporting hybrid editors. He claims that text-oriented commands are not the answer for filling the gaps in structure editors. Instead, he proposes the extension of structure editing commands. He claims that with a sufficiently robust set of structure editing commands, the importance of text-oriented commands is strongly downplayed, if not eliminated.

Cohen [6] also favors pure structure editors and states that the textual viewpoint is never necessary. He contends that three modifications will solve the problems in structure editors without reverting to a textual viewpoint. First, cursor movement should be simplified by viewing the screen as a 2-Dimensional arrangement of nodes. Movement between nodes would be accomplished with the use of natural commands such as up, down, left and right.

The second modification would allow expressions to be input from left to right. This would allow more complex commands to be input than the simple template expansion of most

structure editors.

Third, the transformation of program fragments can be accomplished by an editor which supports matching and instantiation of subtrees. He cites MYTE, under development at Brandeis University, as using this method of fragment modification. In Cohen's mind, these three modifications would fill the gaps of structure editors without the use of text-oriented commands.

In summary, most authors agree that the basic structural viewpoint is an effective and desirable method of program modification. However, the methods for filling the voids of structure editors vary. The hybrid approach (ie. inclusion of text-oriented commands in a structure editor) appears to be the most widely used. This is probably due to its relative ease in implementation. Other methods are certainly under development so, as with all problems of today, we can look to the future for additional innovative solutions.

1.2.2 Resident vs. Source-file Systems Sandewall [12]
describes a residential interactive system to mean that the primary copy of the program resides in the programming system itself. INTERLISP is cited as an example of a residential interactive system. A source-file system is defined as having programs maintained as text files which

are loaded by the programming system. MACLISP is a source-file system. When a resident editor exists but is not well-developed, an intermediate stage is reached. PROLOG is in the intermediate stage, due to the "retract" and "assert" commands, as is LISP 1.6. However, because the capabilities resident in PROLOG and LISP1.6 are not sufficient, most users prefer to use a general-purpose text editor, thus running in source-file mode.

Resident editors are superior to source-file editors because program modules can be modified and debugged without the cost of reloading source-files into the programming environment. However, sufficient capabilities must be included in residential systems to ensure that users do not revert to source-file editors. An example of a useful facility is the ability to write procedure definitions to a file in an input-compatible format. This allows these files to be reread into the programming environment and provides transfer of programs (a procedure may be needed by more than one program), back-up capability, and copy for the user.

1.2.3 Display-Oriented vs. Hard-Copy Editors Display-oriented editors, or full-screen editors, allow the user to view a larger amount of text than hard-copy editors. Cursors allow movement around the displayed area and usually scroll the text as new lines are added. This is in contrast to hard-copy editors which require the user to specify the

lines to be printed. Display-oriented editors are usually preferred by users, especially when viewing large amounts of text.

1.2.4 Residential LISP Editors The resident LISP editors surveyed were LEDIT [10], Franz Lisp [9], EMACS [15], Interlisp [1,18] and ZMACS [7,14]. Text versus structure capabilities were noted as well as display-oriented versus hard copy capabilities. (Display-oriented editors are sometimes called full-screen editors.)

All of the editors are basically structure editors. In addition, some text-oriented commands are present in all. The editors vary in their command structures, but most contain the same basic capabilities. For example, an editor with a more rigorous command structure will allow an edit command to be entered with fewer key strokes than an editor with a limited command structure. Although the capabilities are similar, differences are perceived by the user. While fewer keystrokes are generally desired, the number of commands will be greatly increased. It is a question of personal preference whether fewer keystrokes or fewer commands to remember is more desirable.

EMACS and ZMACS were the only editors surveyed which incorporate full-screen editing support. EMACS is the precursor of ZMACS, and as such can be thought of as a

subset of ZMACS. ZMACS was developed to facilitate editing of English text as well as for preparation, testing and debugging of LISP programs. To support the editing of English text, ZMACS has a larger number and variety of commands than most editors. This includes a full range of mouse manipulations. These extensions make ZMACS by far the most powerful editor surveyed.

1.2.5 Conclusions Basically, three major choices are made when developing or evaluating editors. The first involves structure versus text-oriented commands. Although pure structure editors are far superior to pure text editors, it is widely accepted that some text editing capabilities are needed in most, if not all, structure editors.

The second choice is between resident and source-file editors. A resident editor is of supreme importance in interactive programming environments to save time, effort and resources. However, a resident editor must be rigorous enough to keep users from reverting to source-file editors.

The final choice is between display-oriented (full-screen) editors and hard-copy editors. Display-oriented editors are usually easier to learn in addition to saving time by keeping the current picture displayed at all times. In this way, the user does not need to print the current status of the edited item. However, display-oriented editors are more

costly to implement so are usually not developed when cost is of extreme importance.

Most users would agree that a resident, display-oriented structure editor is most desirable. The development of all these capabilities in an editor should be done simultaneously to ensure efficiency and to reduce duplication of effort. If sufficient resources are not available to include all of these capabilities, choices become necessary. If a single property were to be omitted, the best choice would be the display-oriented capabilities. This would result in a resident structure editor. The inclusion of these two properties in an editor would still fulfill the needs of a user when editing in an interactive programming environment such as PROLOG or LISP. The following chapters describe the development of PEDIT - a resident structure editor for PROLOG. Details for using PEDIT are included in Appendix C, the User's Manual.

2. Chapter Two

2.1 Requirements

The Real-Time LISP Editor [10] was referenced for direction for the requirements of the PROLOG Editor - PEDIT. Since many LISP users are also PROLOG users, the decision was made to retain a maximum number of command abbreviations from the LISP Editor. This is thought to be an important consideration because a user will be able to easily cross between these editors. In general, the editor must be simple enough to ensure that a PROLOG programmer will use it plus it should be versatile enough to compete with an editor external to the PROLOG interpreter.*

The PROLOG editor is also similar to the ED3 editor [16]. It is a structure-oriented text editor, or hybrid editor, in order to allow full editing capabilities of a PROLOG clause. The main thrust of PEDIT is as a structure editor but some text-editing capabilities were provided. A PROLOG clause, or structure as it is called here, usually consists of two parts: a head and a tail separated by ':-' (colon-dash). When a structure has no tail, it is usually referred to as a

* NOTE: Appendix B contains a list of PEDIT commands.

fact.

example: fact(Varlist).

head(Varlist) :- tail.

Any portion of a PROLOG structure can be a compound structure. The editor must be able to address any part of a PROLOG structure, whatever form it takes. This is done by allowing the user to traverse the structure using commands such as right, left, up or down.

For modification of the current structure, commands are provided to find an argument, change an argument, or insert new information either before or after the structure. These commands should be versatile enough to allow the user to easily modify the selected portions of the structure.

The editor should be user-friendly in a few specific areas. First, the editor should provide help to the user in learning how to use the instructions. One feature that is desirable in all systems, no matter how complex, is a facility to provide on-line user assistance. The editor should also allow the user to back out of either single changes or entire edit sessions. An "undo" function is considered an important item, especially in an interactive system such as PROLOG. Last, the editor should allow the user to modify the editor environment. This will save time and effort when a user is involved in lengthy edit sessions.

The top level commands of the editor allow the user to enter and exit the editor. The option exists to save the changes entered within an edit session or to end a session without a save. A function also exists to display the edit commands with a brief explanation on their usage.

A set of commands exist to modify the editor environment. A command is provided to repeat the last executed command. The option also exists to define new editor commands. Any new commands will remain in the PROLOG data base for the duration of the PROLOG session. A command to undo the effects of the previous editor command is also provided by PEDIT.

The user can enter multiple editor commands per line. These are executed in the order that they appear. If an error occurs in the string, the rest of the line is abandoned.

Buffers are also available for storage and movement of the current structure. Commands exist to display the buffer names and their contents.

The implementation of both structure editor commands and text editor commands makes the PROLOG editor versatile. The addition of the commands to modify the editor environment further increases the power of the editor, thus ensuring its use. The requirements defined here will ensure an easily usable editor that is user-friendly and provides powerful

edit capabilities. Chapter Three outlines the design process for implementing the requirements outlined here.

3. Chapter Three

3.1 Design

The implementation of the PEDIT editor is modular in structure. This is a self-imposed requirement as well as a PROLOG-influenced requirement. It is self-imposed for ease of maintainability. As Clocksin and Young stated, "If the program has been made modular - and PROLOG makes this easy to do - then there is usually no need to rewrite existing parts of the program when modifications are required."¹ A structured implementation is PROLOG-influenced because PROLOG is more easily readable and implemented in a modular format. To again quote Clocksin and Young, "The programming style implied by PROLOG makes structured programming much more convenient than with Pascal, for example."²

The main program 'pedit' calls the routine 'edit' to execute the editor. This routine retrieves the desired clause (procedure) using the subroutine 'get_routine'. The procedure occurrences are then returned and passed to the subroutine 'randex'. The modular structure of PEDIT allows for easy addition of commands. See Chapter Five for details

1. Clocksin, Young, p. 16.

2. *ibid*, p. 5.

on adding commands.

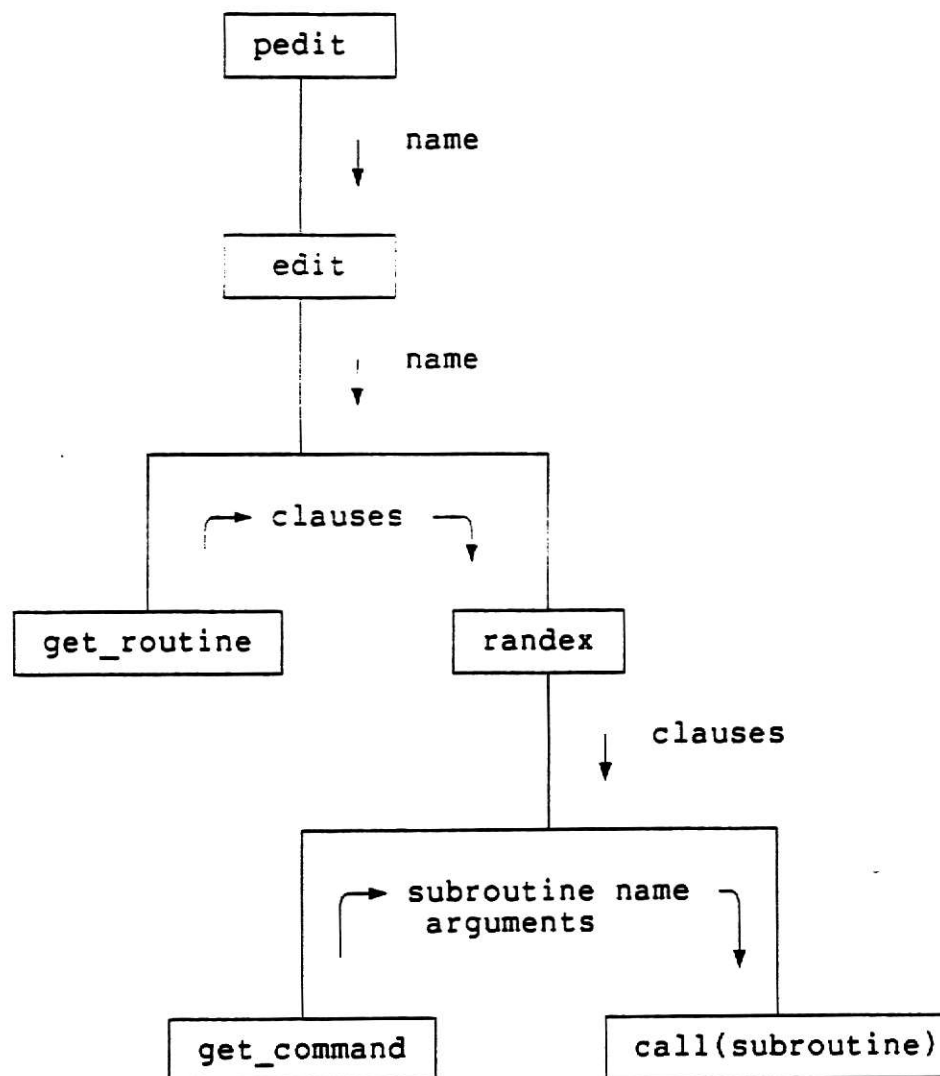


Figure 1. Program Flow

The method of traversing PROLOG structures was an integral part of the design of PEDIT. A format for storing structures inside the editor was required. Many methods

were examined before a choice was made. Most were variations of a common theme, all centering on the idea of treating a PROLOG structure as a tree.

All levels of a structure have basically two parts: the functor and the argument list. Example: functor(arg1, arg2). If a structure is drawn as a tree with the functor as the root and the arguments as nodes, the method of traversal becomes apparent.

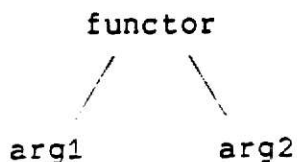


Figure 2. Structure Tree

3.1.1 Traversal Commands To move from the functor to the argument list, the direction is 'Down'.* Movement between nodes, or arguments, is 'Left' or 'Right'. Movement from the argument list to the functor is 'Up'.

A clause can be thought of as a structure with the functor ':-' (pronounced 'if') and argument list (Head, Body). When

* Commands are written in this paper showing the complete name; however, only the upper case portion of the name should be entered as the command. For example, a command written as 'InsertBefore' should be entered into the editor as 'ib'.

the clause is a fact, Body is 'true'. Figures 3 and 4 depict the trees for a rule and a fact, respectively.

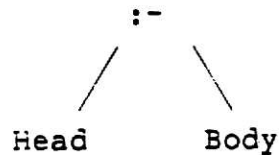


Figure 3. Rule Tree

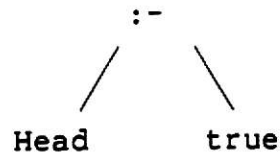


Figure 4. Fact Tree

Each node in the tree can be a structure, so multiple levels are created. When there are multiple goals in the Body, the goals are considered bound together by the functor `' , '` [5]. Figures 5 through 7 show the mapping of a PROLOG structure to a tree structure to an internal PEDIT storage format.

```

functor1(Arg1, Arg2) :- goal1(X1,X2),
                        goal2(B1).

```

Figure 5. PROLOG Structure

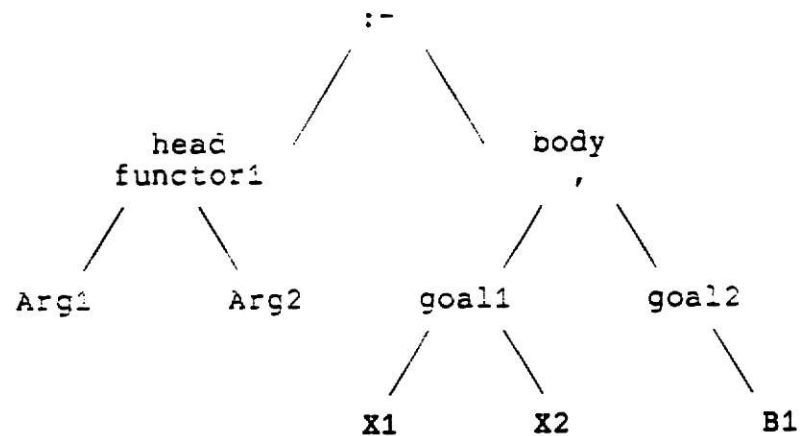


Figure 6. Rule Structure

```

:- (functor1 (Arg1, Arg2),
    ,      (goal1 (X1, X2),   goal2 (B1)      )    )

```

Figure 7. Internal PEDIT Format

In this way, all movement required to reach various positions in a PROLOG clause could be accomplished by using four commands: 'Up', 'Down', 'Left', and 'Right'.*

Limitation to these four commands would, of course, require a user to mentally translate a clause to its equivalent tree format when using the editor. This would not usually present a problem; however, in some cases, the translation from clause to tree is not immediately apparent. This is especially true when attempting to reach the head or body of a rule or when traversing the goals in the body of a rule.

For example, traversal from the highest structure level 'Head :- Body.' to Body would be accomplished by using 'Down' followed by 'Right'. Since this is not a usual way of viewing such a traversal, confusion would usually occur. To minimize confusion in structure traversal, some additional traversal commands were added.

To move from the highest structure level in a clause to its arguments, the commands 'Head' and 'Body' were added. The resulting edit list would be the entire Head or the entire Body of the clause. At this level, the entire edit list

* NOTE: All traversal commands, except Head, Body, Start and TOP, can have repetition arguments of either a number or * to denote the maximum number.

could be replaced or modified. To modify only the argument list of the Head, the user must first travel 'Down'.

Commands were created to traverse multiple goals because of their method of internal storage. Figure 8 illustrates the mapping of the body 'goal1, goal2, goal3, goal4, etc.' to its associated tree structure.

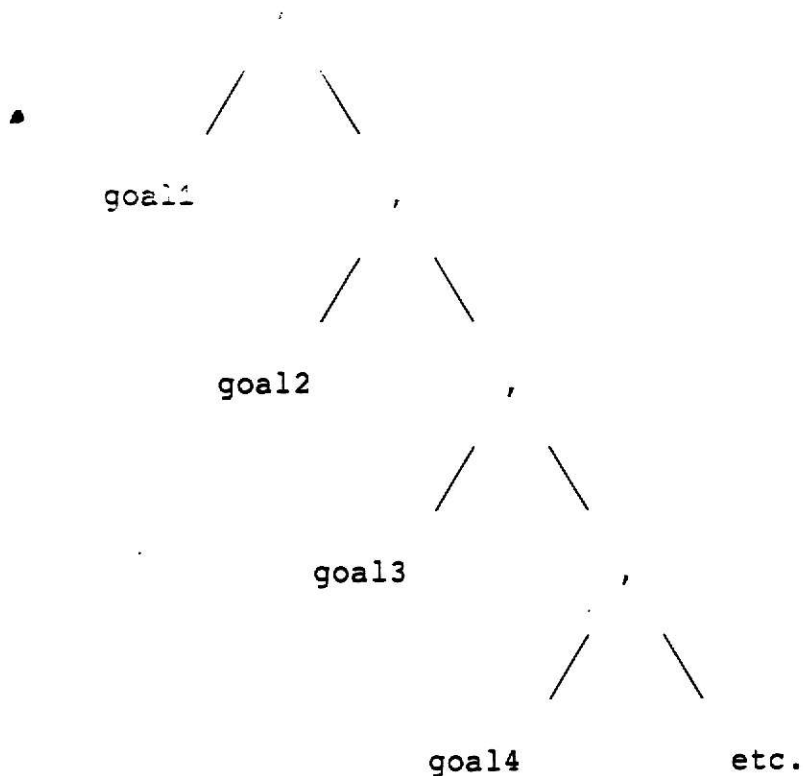


Figure 8. Goal Tree

Thus, to travel from goal1 to goal3 the user would enter 'Right', 'Down', 'Right', 'Down'. To facilitate movement of this kind, the commands 'NextGoal' and 'PreviousGoal' were

added. The addition of the commands 'Head', 'Body', 'NextGoal', and 'PreviousGoal' allows the traversal of a clause in a more logical manner.

Since each occurrence of a procedure produces a tree, travel between trees must also be possible. This is accomplished by the inclusion of the commands 'NextClause' and 'PreviousClause'. The trees are stored in a list, thus producing:

```
[ :- (head, body1), :- (head, body2), . . . ]
```

Each of the traversal commands only allow movement if the appropriate format is present. For example, to use Head or Body, the functor of the current structure level must be ':-'. To travel 'Down', the current level must be a structure, not a list. To travel 'Right', the current level must be a list, such as an argument list. To use 'Next' or 'Previous', the current edit list must be a clause (i.e. functor equals ':-'). Appropriate checks are also made before a "return command" (PreviousClause, Start, PreviousGoal, Up or Left) is executed.

The commands 'TOP' and 'Start' are the exceptions of the traversal commands. The 'TOP' command recurses to the beginning of the first procedure occurrence by executing the appropriate traversal commands. The 'Start' command

recurses to the beginning of the current procedure occurrence. These commands are accomplished by recording all the traversal commands executed during the edit session for a clause. This list of commands is then used to perform the traversal to the beginning of the clauses.

Once the desired structure level is reached, various commands can be issued to modify the structure.

3.1.2 Modification Commands The original intent for the modification commands was to incorporate both structure editing and text editing capabilities in PEDIT. This was described in earlier chapters as a hybrid editor.

As the design of PEDIT progressed, it was determined that structure editing capabilities were more applicable than text editing capabilities when editing PROLOG clauses. This was mainly due to the inability to extract variable names from the data base. Because of this, the text editing needs are narrowed down to editing functor and atom names. This essentially cuts the amount of text editing in half.

The majority of changes required when debugging PROLOG clauses seem to be structural. For example, clauses and goals need to be moved, changed and added. Structure editing commands aid in maintaining the structural integrity of clauses when making these types of modifications. With this in mind, the design proceeded using mainly structure editing

commands.

The first obvious modification needs were to change functors, goals and arguments to a functor. These needs were filled by three commands tailored to those changes. The commands 'CHangeArgument' and 'CHangeGoal' modify positional items and the command 'CHangeFunctor' modifies the functor for the current structure.

A fourth command 'CHange' was added to allow more powerful modifications. This command replaces the current structure with whatever structure is entered. Since structural checks are not performed, this command is meant for careful use. This command plus the 'REPLace' command provide the text editing capabilities which make PEDIT a hybrid editor.

In addition to these commands, the current structure can be removed and replaced using buffers. Commands to examine the buffers are also available to aid in the use of buffers. Again, structural checks are not performed when using buffers so careful use is advised.

The decision was made to limit the capabilities of the search mechanism. The 'Find' command was designed to find a specific occurrence of a clause, not a specific goal or functor. It was felt that since the size of most PROLOG clauses is not usually large, a more powerful search mechanism was not immediately needed.

The next modification function to be designed was the insert mechanism. Separate instructions were designed for inserting either before or after the current structure. The cursor position is unaffected by the insert commands.

Finally, commands were designed to delete structures and move goals. The move command was limited in scope in anticipation of the type of moves needed most. The delete command was designed to delete a specified number of items. If the cursor is at the clause level, clauses will be deleted. If the cursor is at the goal level, goals will be deleted. Otherwise, arguments will be deleted.

The command base was designed to handle a large portion of anticipated modifications. The implementation of PEDIT is described in Chapter Four. Chapter Five outlines some extensions to PEDIT and details the method for extending the PEDIT command base.

4. Chapter Four

4.1 Implementation

PEDIT was developed and tested on a VAX machine running C-PROLOG. PEDIT was written in PROLOG with 450 PROLOG clauses, a total of approximately 60,000 bytes of code.

The implementation of PEDIT is recursive with each structure level having a loop to read commands. At each structure level, the program loops through the command list until the "return command" is found. When a "downward traversal" command is executed, another level of recursion is begun continuing the loop through the command list. Each downward traversal command has an appropriate return command. For example, the return command for 'Down' is 'Up'.

The commands are interactively accepted from the user and are inserted onto the command list. The commands must be entered in all lower case and must be spelled correctly. As commands are recognized, the appropriate command structure is anticipated. If the correct command structure is not entered, that command plus the rest of the line are ignored. When a "new line" is found, the list becomes available for execution.

As the command list is executed, the commands are removed from the list. When a command is unable to execute

successfully, the remaining elements of the command list are deleted. When the program finds a null command list, the user is prompted for further input. The commands read in from the user are then used to create a new command list. Each member of the command list contains the procedure name and the argument list. This argument list is passed into the appropriate command procedure at execution time. Figure 9 depicts the flow of data from its input as a command line to the execution of the command procedures. Examples of the argument lists are [*] for a traversal command or [string, delimiter] for a modification command.

Each command subroutine has six parameters. The first two parameters are passed into the subroutine. First is the argument list for the command and second is the clause name currently being edited. The third parameter is the current structure and is passed into the subroutine. The fourth parameter is the returning structure, and may or may not have changed from the structure passed into the subroutine. The fifth parameter is the "undo" structure passed into the subroutine and the sixth parameter is the returning "undo" structure.

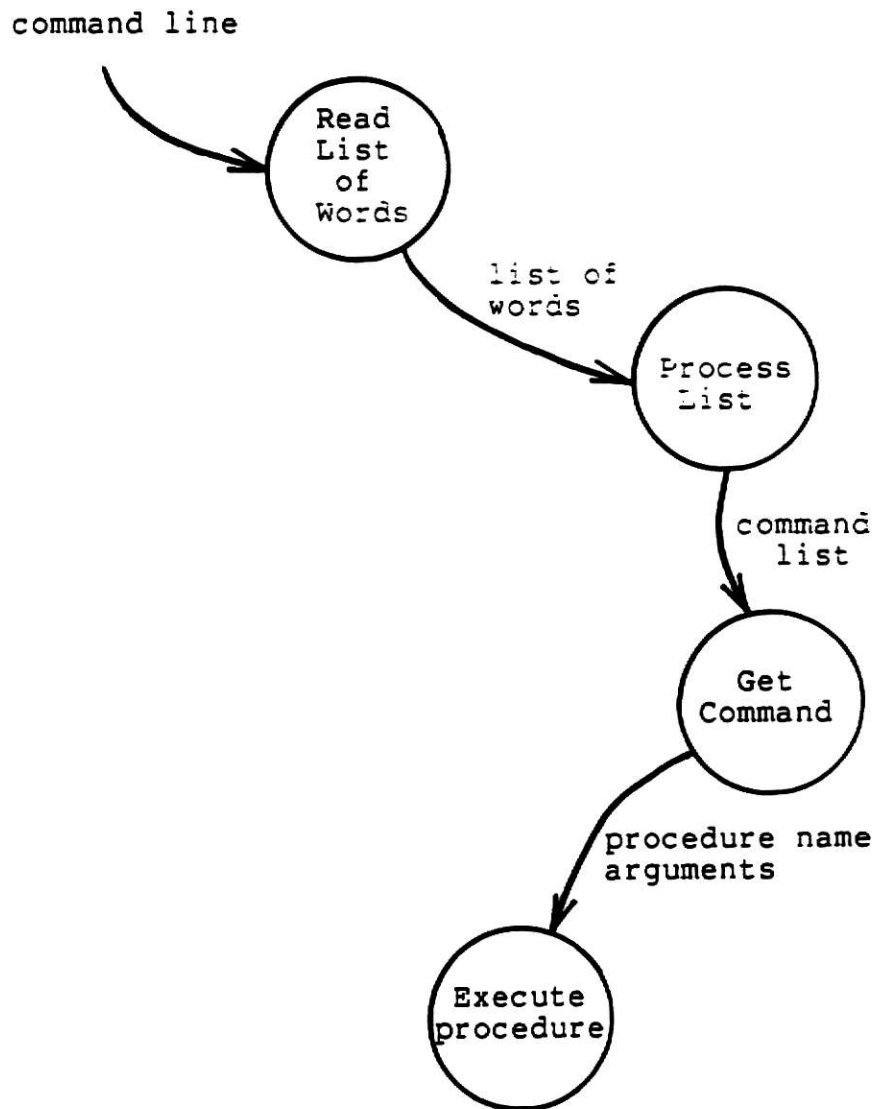


Figure 9. Input Command Flow

As each command is successfully executed, a trail is left behind. A list of all downward traversal commands is kept for each clause name edited. This allows for checks when upward traversal commands are executed, plus it allows for the execution of a "reverse traversal" as is the case for the 'TOP' and 'Start' commands.

The trail also includes the undo list which is updated for all appropriate commands successfully executed. This list consists of the last command executed. Commands such as 'Type', 'RePeaT', 'DISPLAY' and 'CONTENTS' do not change the undo list so they can not be "undone". Also, the 'UNdo' command can not be "undone".

When the 'UNdo' command is executed, the last command is first checked. If it was a traversal command, the opposite traversal command is inserted onto the command list.* For all other commands which can be "undone", the structure passed in is replaced by the last structure encountered. This would apply for the commands which modify the structures, such as 'CHange' and 'DElete'.

The 'UNdo' command will only reverse a single occurrence of

* NOTE: An attempt to undo a 'Start' command will result in the 'Head' command. An attempt to undo a 'TOP' command will result in the 'NextClause' command.

a traversal command. In other words, if a 'Right 5' were undone, the results would be a single 'Left' command. The same is true for traversal commands with the argument '*'.

An edit session can be ended in three ways: QUIT, FILE or END. A 'QUIT' command aborts the current edit session. If changes have been made but not saved, the user is prompted for verification. If 'QUIT' is again entered, the session is aborted; otherwise, the 'FILE' command is executed.

The 'FILE' command replaces all predicate occurrences in the PROLOG data base with the edited structures. The command recurses to the start of the first procedure occurrence before performing its functions. This is done by executing the 'TOP' command. The command then asks the user if the structures are to be stored in a file. If the user answers 'yes', the structures are stored in the filename supplied by the user. Otherwise, the command is exited.

If, for some reason, the structures cannot be added to the data base, an error message is given. The structure is then saved in the file 'peditjunk'. This allows the user to examine the structure at a later time without the loss of the edited version. For this error, the pre-edit session version of the current clause is left intact in the PROLOG data base.

Since variable names can not be retrieved from the PROLOG data base, the structures written to the file will have internal variable representation (ex. _23). The user can later do global edits on the file to reassign the pre-PROLOG session variable names.

An 'END' command prompts the user for 'QUIT or FILE'. The appropriate command is then executed. If 'FILE' is entered, a 'QUIT' command is executed after the 'FILE' has completed.

4.2 Testing

Due to the structure of PROLOG clauses, the procedures were easily unit-tested. This allowed all of the modules to be verified before the program was system tested. Once the internal format for the clauses was established and tested, the driving routines for PEDIT were implemented and tested. Finally, the command procedures were tested. The first group of commands to be tested were the traversal commands. The general edit commands were tested next. Finally, the modification and special editor commands were tested.

Integration tests were routinely performed as the unit-tested routines were incorporated into the system. Finally, thorough system tests were performed to ensure that all parts integrally worked together.

5. Chapter Five

5.1 Conclusions

Efforts were made to provide the capability of modifying all portions of a PROLOG clause. However, due to some inherent deficiencies, such as the inability to retrieve variable names, PEDIT does not satisfy all desirable elements of an editor. This, of course, would be a natural extension if and when those deficiencies are removed.

Though attempts were made to model PEDIT after LEDIT - The Real-Time LISP Editor [10], the differences between PROLOG and LISP necessitated differences in editors. As the design progressed, different methods of traversing the structures became apparent. Since a PROLOG clause is more variable in structure, a more rigorous set of traversal commands was implemented.

The modification commands were also different in implementation. Since variable names can not be extracted, positional modifications were needed in PEDIT. Also, some commands used in LEDIT were not applicable for modifying PROLOG clauses.

As was mentioned in Chapter Three, although the original intent was to implement PEDIT as a hybrid editor, the text editing capabilities were found to be mainly unnecessary.

Most of the editing commands that were implemented were structure editing in nature. The requirement of a hybrid editor was met in spirit if not in letter.

PEDIT succeeds in many of the requirements. First, it succeeds in the requirement of addressing any part of a PROLOG clause. It also fulfills the user-friendly requirement. For example, assistance is provided in the 'HELP' command and the 'UNdo' command provides the user an opportunity to correct inadvertent errors. Also, the user is protected from accidentally exiting the editor without saving the changes made by means of an additional prompt.

5.2 Extensions

It can be safely said that no editor has ever fulfilled all the requirements for every user. In this respect, PEDIT is no different. With this in mind, PEDIT was designed to facilitate extensions by future users. A useful extension would be a more powerful "Find" mechanism. This could include the ability to search between clauses as well as within clauses. The following paragraphs describe the necessary ingredients for incorporating new commands.

First, the mapping of an input command string to its associated procedure and argument list must be added. This is accomplished by adding occurrence(s) to the 'process_command' procedure. This procedure receives words

from the input string and parses the proper number of arguments. Some rudimentary syntax checks are also performed. The procedure name and the appropriate arguments are then added to the command list.

All new command procedures should call the procedure 'leave_trail' to provide the markers needed for the 'UNdo' command. When appropriate, the command name should be added to the 'invalid_undo' list. If the command is a modification command, the procedure name should be added to the 'modification_command' list. If the command is a traversal command, the procedure name should be added to the 'traversal_command' list. This list contains three parameters: the name of the command in the first position, either 'up' or 'down' in the second position dependent on whether the command is a downward or upward traversal command, and the reverse command procedure in the third position which is used by the 'UNdo', 'TOP', and 'Start' commands.

Each command procedure has six parameters. These have been described in Chapter Four. If the new command creates another recursive level by calling the procedure 'randex', the command to exit that level should be passed to 'randex' as the first argument. When an error occurs, the procedure 'error' should be called. This procedure has two parameters: the procedure name and the current structure.

This procedure prints a simple error message and "pretty prints" the current structure. The command list is then purged to avoid compounding the error. If an additional message is desired, it can be printed either before or after the call to 'error'.

By following these instructions, a user should be able to extend the capabilities of PEDIT. This should ensure the continued use of PEDIT as a useful PROLOG-based editor.

BIBLIOGRAPHY

- [1] Barstow, David R. "A Display-Oriented Editor for Interlisp." Interactive Programming Environments. McGraw-Hill, Inc. 1984.
- [2] Barstow David R. Shrobe, Howard E. Sandewall, Erik. Interactive Programming Environments. McGraw-Hill, Inc. 1984.
- [3] Brown, P. J. Writing Interactive Compilers and Interpreters. John Wiley & Sons. New York, New York. 1979.
- [4] Clocksin, W. F. Young, J. D. "Introduction to PROLOG, a 'Fifth-Generation' Language." ComputerWorld. USA. Volume 17. Number 31. P. 1D1-16. August 1, 1983.
- [5] Clocksin, W. F. Mellish, C. S. Programming in PROLOG. Springer-Verlag. 1981.
- [6] Cohen, E. "Text-Oriented Structure Commands for Structure Editors." SIGPLAN Notices. Volume 17. Number 11. November, 1982.
- [7] Cohen, Meryl. Ingria, Robert. "Introduction to the Lambda: A Programmer's Guide to Getting Started." LISP Machine Inc. Los Angeles, CA. August, 1984.
- [8] Donzeau-Gouge, Veronique. Huet, Gerard. Kahn, Gilles. Lang, Bernard. "Programming Environments Based on Structured Editors: the MENTOR Experience." Interactive Programming Environments. McGraw-Hill, Inc. 1984.
- [9] Foderaro, John K. Sklower, Keith L. The FRANZ LISP Manual. Berkeley, CA. April, 1982.
- [10] Goodman, Jana Taylor. The Real-Time Lisp Editor. Manhattan, KN. 1982.
- [11] Ledgard, Henry. Singer, Andrew. Whiteside, John. Directions in Human Factors for Interactive Systems. Springer-Verlag. New York, New York. 1979.
- [12] Sandewall, Erik. "Programming in an Interactive Environment: the LISP Experience." Interactive Programming Environments. McGraw-Hill, Inc. 1984. PP. 31-80.

BIBLIOGRAPHY

- [13] Shani, Uri. "Should Program Editors not Abandon Text Oriented Commands?" SIGPLAN Notices. Volume 18. Number 1. January 1983.
- [14] Smith, Sarah. "ZMACS Introductory Manual." LISP Machine Inc. Los Angeles, CA. 1984.
- [15] Stallman, Richard M. "EMACS: The Extensible, Customizable, Self-Documenting Display Editor." Proceedings of SIGPLAN/SIGOA Symposium on Text Manipulation. June 8-10, 1981. Portland, ORE. P. 147.
- [16] Stromfors, Ola. Jonesjo, Lennart. "The Implementation and Experiences of a Structure-Oriented Text Editor." Proceedings of SIGPLAN/SIGOA Symposium on Text Manipulation. June 8-10, 1981. Portland, ORE. P. 22.
- [17] Teitelbaum, Tim. Reps, Thomas. "The Cornell Program Synthesizer: A Syntax-Directed Programming Environment." Communications of the ACM. Volume 24. Number 9. September, 1981.
- [18] Teitelman, Warren. Masinter, Larry. "The Interlisp Programming Environment." Interactive Programming Environments. McGraw-Hill, Inc. 1984. PP. 83-96.
- [19] Waters, Richard C. "Program Editors Should Not Abandon Text Oriented Commands." SIGPLAN Notices. Volume 17. Number 7. July, 1982.
- [20] Wilander, Jerker. "An Interactive Programming System for Pascal." Interactive Programming Environments. McGraw-Hill, Inc. 1984.

APPENDIX A

Glossary

argument	Objects related to a functor. Example: functor(argument1, argument2)
body	Set of calls to other clauses. See format for rule.
clause	A PROLOG statement.
database	Sequence of modules that have been read in.
fact	Clause without a body. Format - 'head.'
functor	Clause name; also called a predicate.
goal	Argument in the body of a rule.
head	See rule and fact for format.
module	Sequence of routines; every routine is contained inside some module.
predicate	Clause name; also called a functor.
program	Sequence of clauses.
routine	Sequence of clauses each with the same predicate. Also called a procedure.
rule	Full form of a clause. Also known as an implication. Format - 'head :- body.'
structure	Any portion of a rule.

APPENDIX B

PEDIT Commands

General Edit Commands

QUIT	Abort this edit session
FILE	Save this edit session
END	End this edit session
HELP	Display edit commands
Type	Display current edit list

Traversal Commands

TOP	Move the cursor to the beginning of the first procedure occurrence.
NextClause	Move the cursor to the next procedure occurrence
PreviousClause	Move the cursor to the previous procedure occurrence
Head	Move the cursor to the head of the current procedure occurrence
Body	Move the cursor to the body of the current procedure occurrence
Start	Move the cursor to the start of the current procedure occurrence
NextGoal	Move the cursor to the next goal in the body
PreviousGoal	Move the cursor to the previous goal in the body
Down	Move the cursor to a lower structure level
Up	Move the cursor to a higher structure level
Right	Move the cursor right in the structure
Left	Move the cursor left in the structure

APPENDIX B

Modification Commands

Find	Find a given clause
DELeTe	Delete structure(s)
InsertAfter	Insert after the current structure
InsertBefore	Insert before the current structure
MoVe	Move structure(s)
CoPy	Copy structure(s)
CHaNgE	Change current structure
CHaNgEARGument	Change argument of current structure
CHaNgEFUNctor	Change functor name of current structure
CHaNgEGOAL	Change goal of current structure
REMove	Remove current structure and place in buffer
REPLace	Replace current structure with contents of buffer

Special Editor Commands

UNdo	Undo last editor command
BUFFER	Create a buffer
DISPLAY	Display all buffer names
CONTENTS	Display contents of a given buffer
CHaNgEDELIteR	Change current delimiter
RePeaT	Repeat the last command
DEFineComManD	Define a command for editor use
DISPLayComManDs	Display the names of all new commands
COMMAND	Display the list of commands associated with a new command

APPENDIX C

USER'S MANUAL

Introduction - How to Use PEDIT

This manual describes the use of a real-time editor for PROLOG. The editor will enable the leaving the PROLOG interpreter. Edited clauses may replace clauses in the PROLOG data base and may be stored in files. The editor is a structure editor with facilities to edit all portions of a PROLOG structure.

All commands must be spelled correctly and must be entered in all lower case. Multiple commands may be entered on an input line. Items are separated by single or multiple blanks and the input line is ended by a carriage return. As the input line is parsed, rudimentary syntax checks are made. If a command is not input in the proper format, the input line from that point forward is deleted. Commands may not be broken across lines.

Commands are written in this manual showing the complete name; however, only the upper case portion of the name should be entered as the command. For example, a command written as 'InsertBefore' should be entered into the editor as 'ib'.

Due to the internal storage of strings, goals such as 'write('error')' are retrieved as 'write(error)'. To avoid syntax errors when the clause is replaced in the data base, strings such as these should be restored while editing the clause. When strings are entered 'write("error")', no problem occurs.

Since spaces are used to separate 'words', care should be used when typing in new structures. For example, while 'arg1, arg2' is a legitimate entry to the PROLOG interpreter, the items would be separated when parsed by PEDIT. In this case, the proper entry would be 'arg1,arg2'.

Variables appear in the internal format of PROLOG (_23). This format can be input to the modification commands (ex. chang 1 _23) or a new variable name can be identified (ex. chang 1 \bar{X}). The names entered via PEDIT commands create new internal mappings when the clauses are added to the PROLOG data base. These mappings will not match any mappings performed in previous assertions.

Items added via PEDIT are stored as strings until added to the data base. Because of this, when a functor and its

APPENDIX C

arguments are entered as one item (ex. functor(arg1, arg2)), this should be considered to be a single entity. These items can not be accessed in the same manner as those retrieved from the data base. For example, the 'Down' command can not be performed to reach the arguments.

NOTE: If the edit session terminates abnormally, execute <cleanup.>. This clause will remove any remnants of the previous edit session(s) and will allow a new edit session to be started.

APPENDIX C

Entering and Leaving PEDIT

To enter the editor from the PROLOG interpreter environment type:

```
pedit(clausename).
```

The 'clausename' must be structurally identical to the clause to be edited. In other words, if arguments are present, the correct number of arguments must be entered in order to retrieve the proper clauses. An example would be:

```
pedit(clausename(Arg1,Arg2,...)).
```

Arg1, Arg2, etc. should be specified as variable names to allow internal instantiation to the argument names in the PROLOG data base. Any variable names can be entered, except the anonymous variable '_' because internal instantiation is stifled.

To exit the editor, one of the following commands should be entered: END, QUIT or FILE. These commands are explained on their appropriate manual pages.

APPENDIX C

QUIT

quit

The 'QUIT' command aborts the current edit session. If changes have been made to the clause but have not been saved, the user is prompted for verification. Enter either 'quit.' or 'save.'. If 'quit.' is entered, the session is aborted; otherwise the 'FILE' command is executed. There are no arguments for this command.

APPENDIX C

FILE

file

The 'FILE' command replaces all predicate occurrences in the PROLOG data base with the edited structures. The command then asks the user if the structures are to be stored in a file. Enter either 'yes.' or 'no.'. If 'yes.' is entered, the structures are stored in the filename supplied in the format 'filename.'. Otherwise, the command is exited. The 'FILE' command ends the current edit session.

If, for some reason, the edited structures can not be properly formatted for input into the data base, an error message is printed and the structures are saved in the file 'peditjunk'. The data base version of the clause will remain intact in the pre-edit session status.

There are no arguments for this command.

APPENDIX C

END

end

The 'END' command prompts the user 'quit or save?'. The response should be formatted either 'quit.' or 'save.'. If 'save.' is entered, the 'FILE' command is executed; otherwise, the 'QUIT' command is executed. There are no arguments for this command.

APPENDIX C

HELP

`help <command name>`

The 'HELP' command displays an explanation of the syntax and the function of the command name entered. To display a list of all valid commands, type 'help help'. The abbreviated command name is used for input. For example, 'help pg' would provide the explanation for the traversal command 'PreviousGoal'.

APPENDIX C

Type

t

The 'Type' command causes the current structure to be displayed in 'pretty print' format. The current structure remains the same. There are no arguments for this command.

APPENDIX C

TOP

top

The 'TOP' command moves the cursor to the beginning of the first procedure occurrence. The result will be the entire PROLOG procedure. There are no arguments for this command.

EXAMPLE:

The current procedure is thisclause(arg1).
 thisclause(arg2) :- goal1, goal2.
 thisclause(arg3) :- goal3, goal4.
 thisclause(arg4) :- goal5.

The current structure is goal3, goal4

==> top
 results in thisclause(arg1).
 thisclause(arg2) :- goal1, goal2.
 thisclause(arg3) :- goal3, goal4.
 thisclause(arg4) :- goal5.

NOTE: This command will be automatically issued before a QUIT or FILE command is executed.

APPENDIX C

NextClause

nc or nc <num> or nc *

The command 'NextClause' moves the cursor to the start of the next procedure occurrence. When '*' is entered, the maximum allowable moves are made. When 'num' is entered, that number of moves are made. When no argument is entered, only 1 move is made.

EXAMPLE:

The current structure is thisclause(arg1).
 thisclause(arg2) :- goal1, goal2.
 thisclause(arg3) :- goal3, goal4.
 thisclause(arg4) :- goal5.

==> nc 1
 results in thisclause(arg2) :- goal1, goal2.

==> nc *
 results in thisclause(arg4) :- goal5.
 2 repetition(s)

If the editor can not move the cursor the number of moves specified, the editor will simply move the cursor to the last procedure occurrence. A message will be printed telling how many moves were completed.

EXAMPLE:

The current structure is thisclause(arg1).
 thisclause(arg2) :- goal1, goal2.
 thisclause(arg3) :- goal3, goal4.
 thisclause(arg4) :- goal5.

==> nc 5
 results in thisclause(arg4) :- goal5.
 3 repetition(s)

NOTE: A message is also printed telling how many moves were completed when the argument is '*'.

APPENDIX C

PreviousClause

`pc or pc <num> or pc *`

The command 'PreviousClause' moves the cursor to the start of the previous procedure occurrence. When '*' is entered, the maximum allowable moves are made. When 'num' is entered, that number of moves are made. When no argument is entered, only 1 move is made.

EXAMPLE:

The current procedure is `thisclause(arg1).`
 `thisclause(arg2) :- goal1, goal2.`
 `thisclause(arg3) :- goal3, goal4.`
 `thisclause(arg4) :- goal5.`

The current structure is `thisclause(arg3) :- goal3, goal4.`
 `thisclause(arg4) :- goal5.`

`==> pc 1`
 `results in` `thisclause(arg2) :- goal1, goal2.`
 `thisclause(arg3) :- goal3, goal4.`
 `thisclause(arg4) :- goal5.`

`==> pc *`
 `results in` `thisclause(arg1).`
 `thisclause(arg2) :- goal1, goal2.`
 `thisclause(arg3) :- goal3, goal4.`
 `thisclause(arg4) :- goal5.`
 `1 repetition(s)`

If the editor can not move the cursor the number of moves specified, the editor will simply move the cursor to the first procedure occurrence. A message will be printed telling how many moves were completed.

- continued -

APPENDIX C

EXAMPLE:

The current structure is thisclause(arg3) :- goal3, goal4.
 thisclause(arg4) :- goal5.

==> pc 5
 results in thisclause(arg1).
 thisclause(arg2) :- goal1, goal2.
 thisclause(arg3) :- goal3, goal4.
 thisclause(arg4) :- goal5.
 2 repetition(s)

NOTE: A message is also printed telling how many moves were completed when the argument is '*'.
 2 repetition(s)

APPENDIX C

Head

h

The 'Head' command moves the cursor to the head of the current procedure occurrence. There are no arguments for this command.

EXAMPLE:

The current structure is thisclause(arg2) :- goal1, goal2.

```
==> h
      results in               thisclause(arg2)
```

APPENDIX C

Body

b

The 'Body' command moves the cursor to the body of the current procedure occurrence. The result is the entire list of goals in the body. For facts, the result is 'true'. There are no arguments for this command.

EXAMPLE 1:

The current structure is thisclause(arg1).

```
==> b
      results in               true
```

EXAMPLE 2:

The current structure is thisclause(arg2) :- goal1, goal2.

```
==> b
      results in               goal1, goal2
```

NOTE: To manipulate the first goal in the body immediately after the command 'Body' has been executed, the command 'Down' should be entered. To reach other goals in the body, the 'NextGoal' command should be used.

Start

s

The 'Start' command moves the cursor to the start of the current procedure occurrence. The command will recurse to the start no matter where the cursor is currently placed. There are no arguments for this command.

EXAMPLE:

The current procedure is thisclause(arg3) :-
 goal3, goal4(goal_arg).

The current structure is goal_arg

==> s
 results in thisclause(arg3) :-
 goal3, goal4(goal_arg).

The current structure is goal3, goal4(goal_arg)

==> s
 results in thisclause(arg3) :-
 goal3, goal4(goal_arg).

NOTE: This command must be entered in order to reach the level where the commands 'NextClause' and 'PreviousClause' may be entered.

APPENDIX C

NextGoal

ng or ng <num> or ng *

The command 'NextGoal' moves the cursor to the next goal in the body of the current procedure occurrence. When '*' is entered, the maximum allowable moves are made. When 'num' is entered, that number of moves are made. When no argument is entered, only 1 move is made.

EXAMPLE:

The current procedure is thisclause(arg3) :-
 goal3, goal4, goal5, goal6.

The current structure is goal3, goal4, goal5, goal6

==> ng 1
 results in goal4, goal5, goal6

==> ng *
 results in goal6
 2 repetition(s)

NOTE: To edit goals, the user must travel 'Down' to reach a point where editing may occur.

If the editor can not move the cursor the number of moves specified, the editor will simply move the cursor to the last goal. A message will be printed telling how many moves were completed.

EXAMPLE:

The current structure is goal3, goal4, goal5, goal6

==> ng 5
 results in goal6
 3 repetition(s)

NOTE: A message is also printed telling how many moves were completed when the argument is '*'.

APPENDIX C

PreviousGoal

pg or pg <num> or pg *

The command 'PreviousGoal' moves the cursor to the previous goal in the body of the current procedure occurrence. When '*' is entered, the maximum allowable moves are made. When 'num' is entered, that number of moves are made. When no argument is entered, only 1 move is made.

EXAMPLE:

The current procedure is thisclause(arg3) :-
 goal3, goal4, goal5, goal6.

The current structure is goal6

==> pg 1
 results in goal5, goal6

==> pg *
 results in goal3, goal4, goal5, goal6
 2 repetition(s)

NOTE: To edit goals, the user must travel 'Down' to reach a point where editing may occur.

If the editor can not move the cursor the number of moves specified, the editor will simply move the cursor to the first goal. A message will be printed telling how many moves were completed.

EXAMPLE:

The current structure is goal6

==> pg 5
 results in goal3, goal4, goal5, goal6
 3 repetition(s)

NOTE: A message is also printed telling how many moves were completed when the argument is '*'.

APPENDIX C

Down

d or d <num> or d *

The command 'Down' moves the cursor to a lower structure level. When '*' is entered, the maximum allowable moves are made. When 'num' is entered, that number of moves are made. When no argument is entered, only 1 move is made.

EXAMPLE 1:

The current structure is goal3(goal4(thisarg))

```
==> d 2
      results in               thisarg
```

EXAMPLE 2:

The current structure is goal1(goal2(level2(lastlevel)))

```
==> d *
      results in               lastlevel
                               3 repetition(s)
```

If the editor can not move the cursor the number of moves specified, the editor will simply move the cursor to the lowest level. A message will be printed telling how many moves were completed.

EXAMPLE:

The current structure is goal3(goal4(thisarg))

```
==> d 5
      results in               thisarg
                               2 repetition(s)
```

NOTE: A message is also printed telling how many moves were completed when the argument is '*'.

APPENDIX C

Up

u or u <num> or u *

The command 'Up' moves the cursor to a higher structure level. When '*' is entered, the maximum allowable moves are made. When 'num' is entered, that number of moves are made. When no argument is entered, only 1 move is made.

EXAMPLE 1:

The current body is goal3(goal4(thisarg))

The current structure is thisarg

```
==> u 1
      results in                   goal4( thisarg )
```

EXAMPLE 2:

The current structure is thisarg

```
==> u *
      results in                   goal3( goal4( thisarg ) )
                                   2 repetition(s)
```

If the editor can not move the cursor the number of moves specified, the editor will simply move the cursor to the highest level. A message will be printed telling how many moves were completed.

EXAMPLE:

The current body is goal1(goal2(level2(lastlevel)))

The current structure is lastlevel

```
==> u 5
      results in                   goal1(goal2(level2(lastlevel)))
                                   3 repetition(s)
```

NOTE: A message is also printed telling how many moves were completed when the argument is '*'.

APPENDIX C

Right

r or r <num> or r *

The command 'Right' moves the cursor to right in the structure. When '*' is entered, the maximum allowable moves are made. When 'num' is entered, that number of moves are made. When no argument is entered, only 1 move is made.

EXAMPLE 1:

The current structure is arg1, arg2, arg3

```
==> r 2
      results in                    arg3
```

EXAMPLE 2:

The current structure is arg1, arg2, arg3, arg4, arg5

```
==> r *
      results in                    arg5
                                   4 repetition(s)
```

If the editor can not move the cursor the number of moves specified, the editor will simply move the cursor as far right as possible. A message will be printed telling how many moves were completed.

EXAMPLE:

The current structure is arg1, arg2, arg3

```
==> r 5
      results in                    arg3
                                   2 repetition(s)
```

NOTE: A message is also printed telling how many moves were completed when the argument is '*'.

Also, due to the internal storage mechanism, 'Right' should not be used to travel among goals. Instead, the command 'NextGoal' is provided.

APPENDIX C

Left

1 or 1 <num> or 1 *

The command 'Left' moves the cursor left in the structure. When '*' is entered, the maximum allowable moves are made. When 'num' is entered, that number of moves are made. When no argument is entered, only 1 move is made.

EXAMPLE:

The current goal is goal(arg1, arg2, arg3, arg4, arg5)

The current structure is arg5

```
==> 1 1
      results in           arg4, arg5
```

```
==> 1 *
      results in           arg1, arg2, arg3, arg4, arg5
                           3 repetition(s)
```

If the editor can not move the cursor the number of moves specified, the editor will simply move the cursor as far left as possible. A message will be printed telling how many moves were completed.

EXAMPLE:

The current goal is goal(arg1, arg2, arg3, arg4, arg5)

The current structure is arg5

```
==> 1 7
      results in           arg1, arg2, arg3, arg4, arg5
                           4 repetition(s)
```

NOTE: A message is also printed telling how many moves were completed when the argument is '*'.

APPENDIX C

Find

f <num> <pattern>

The 'Find' command searches the bodies of the rules for a match of <pattern> in the argument <num>. If no match is found, the cursor is positioned at the last rule. If a match is found, the cursor is positioned at that rule. There should be no blanks within <pattern>. The items being matched must be instantiated.

EXAMPLE:

```
The current structure is  thisclause(arg1).
                           thisclause(arg2) :- goal1, goal2.
                           thisclause(arg3) :- goal3, goal4.
                           thisclause(arg4) :- goal5.

==> f 1 arg3
      results in          thisclause(arg3) :- goal3, goal4.

==> f 1 arg6
      results in          thisclause(arg4) :- goal5.
                           *** rule not found ***
```

NOTE: If the cursor is not positioned at the start of a rule, the command 'Start' is executed before the search begins.

APPENDIX C

DElete

`del or del <num> or del *`

The 'DElete' command deletes either: single/multiple rules, single/multiple goals, or single/multiple items. When '*' is entered, all the items in the current structure are deleted. When 'num' is entered, that number of items are deleted. When no argument is entered, only 1 item is deleted.

If there are fewer items in the current structure than are specified in <num>, all the items are deleted. If the cursor is positioned at the start of a rule, rule(s) are deleted. If the cursor is positioned at a goal in the body of a rule, one or more goals are deleted. If the cursor is positioned at the arguments to a functor one or more arguments are deleted.

EXAMPLE 1:

```
The current structure is  thisclause(arg1).
                          thisclause(arg2) :- goal1, goal2.
                          thisclause(arg3) :- goal3, goal4.
                          thisclause(arg4) :- goal5.
```

```
==> del 2
      results in          thisclause(arg3) :- goal3, goal4.
                          thisclause(arg4) :- goal5.
```

EXAMPLE 2:

```
The current structure is  goal1, goal2, goal3, goal4
```

```
==> del 1
      results in          goal2, goal3, goal4
```

- continued -

APPENDIX C

EXAMPLE 3A:

The current structure is arg1, arg2, arg3, arg4, arg5

```
==> del 3
      results in               arg4, arg5
```

EXAMPLE 3B:

The current structure is arg1, arg2, arg3, arg4, arg5

```
==> del 6
      results in
                               (null list)
```


APPENDIX C

InsertAfter

```
ia  <head>  <goal list>  <delim>
    or
ia  <goal list>  <delim>
    or
ia  <argument list>  <delim>
```

The 'InsertAfter' command inserts either: a single rule, single/multiple goals, or single/multiple items after the current structure.

If the cursor is positioned at the start of a rule, a new rule is inserted. In this case, the first format should be entered.

If the cursor is positioned at a goal in the body of a rule, one or more goals are inserted. In this case, the second format should be entered. (See the 'Body', 'NextGoal', and 'PreviousGoal' commands for information on reaching the body or goals in a rule.)

The goals in the <goal list> should be separated by blanks. Each goal should be followed by a goal functor - a comma or semicolon. These functors are placed between the goals as they are inserted into the body of a rule. (The <goal list> may be omitted for the first format. The resultant rule is a fact (body = 'true').)

If the cursor is positioned at the arguments to a functor one or more arguments are inserted. In this case, the third format should be entered. (See 'Down' for information on reaching the arguments to a functor.). The items in the <argument list> should be separated by blanks.

The <delim> is equal to '/' (slash) when the edit session begins. (See 'CHangeDELIMiter' to change the current delimiter.)

- continued -

APPENDIX C

EXAMPLE 1A:

The current structure is thisclause(arg1) :- goal1, goal2.
 thisclause(arg3) :- goal3, goal4.

==> ia thisclause(arg2) goal2 goal3 /
 results in thisclause(arg1) :- goal1, goal2.
 thisclause(arg2) :- goal2, goal3.
 thisclause(arg3) :- goal3, goal4.

EXAMPLE 1B:

The current structure is thisclause(arg1) :- goal1, goal2.
 thisclause(arg3) :- goal3, goal4.

==> ia thisclause(arg2) true /
 or
 ia thisclause(arg2) /
 results in thisclause(arg1) :- goal1, goal2.
 thisclause(arg2).
 thisclause(arg3) :- goal3, goal4.

EXAMPLE 2:

The current structure is goal1, goal2

==> ia goal1a , goal2a , /
 results in goal1, goal1a, goal2a, goal2

EXAMPLE 3:

The current structure is arg1, arg2

==> ia arg1a arg2a /
 results in arg1, arg1a, arg2a, arg2

APPENDIX C

InsertBefore

```
ib  <head>  <goal list>  <delim>
      or
ib  <goal list>  <delim>
      or
ib  <argument list>  <delim>
```

The 'InsertBefore' command inserts either: a single rule, single/multiple goals, or single/multiple items before the current structure.

If the cursor is positioned at the start of a rule, a new rule is inserted. In this case, the first format should be entered.

If the cursor is positioned at a goal in the body of a rule, one or more goals are inserted. In this case, the second format should be entered. (See the 'Body', 'NextGoal', and 'PreviousGoal' commands for information on reaching the body or goals in a rule.)

The goals in the <goal list> should be separated by blanks. Each goal should be followed by a goal functor - a comma or semicolon. These functors are placed between the goals as they are inserted into the body of a rule. (The <goal list> may be omitted for the first format. The resultant rule is a fact (body = 'true').)

If the cursor is positioned at the arguments to a functor one or more arguments be inserted. In this case, the third format should be entered. (See 'Down' for information on reaching the arguments to a functor.). The items in the <argument list> should be separated by blanks.

The <delim> is equal to '/' (slash) when the edit session begins. (See 'CHangeDELIMiter' to change the current delimiter.)

- continued -

APPENDIX C

EXAMPLE 1A:

The current structure is thisclause(arg2) :- goal2, goal3.
 thisclause(arg3) :- goal3, goal4.

```
==> ib thisclause(arg1) goal1 goal2 /
      results in                    thisclause(arg1) :- goal1, goal2.
                                   thisclause(arg2) :- goal2, goal3.
                                   thisclause(arg3) :- goal3, goal4.
```

EXAMPLE 1B:

The current structure is thisclause(arg2) :- goal2, goal3.
 thisclause(arg3) :- goal3, goal4.

```
==> ib thisclause(arg1) true /
      or
      ib thisclause(arg1) /
      results in                    thisclause(arg1).
                                   thisclause(arg2) :- goal2, goal3.
                                   thisclause(arg3) :- goal3, goal4.
```

EXAMPLE 2:

The current structure is goal1, goal2

```
==> ib goal1a , goal2a , /
      results in                    goal1a, goal2a, goal1, goal2
```

EXAMPLE 3:

The current structure is arg2, arg3

```
==> ib arg1a arg2a /
      results in                    arg1a, arg2a, arg2, arg3
```

APPENDIX C

MoVe

`mv <fromarg> <toarg>`

The 'MoVe' command moves either: a single rule, a single goal, or a single item. If the cursor is positioned at the start of a rule, a rule is moved. If the cursor is positioned at a goal in the body of a rule, a goal is moved. If the cursor is positioned at the arguments to a functor, an argument is moved.

The structure at position <fromarg> is moved behind the structure at position <toarg>. If either of the arguments is higher than the number of items present, an error is flagged and the command is ignored. Zero is a valid value for <toarg>.

EXAMPLE 1:

```
The current structure is  thisclause(arg1).
                           thisclause(arg2) :- goal1, goal2.
                           thisclause(arg3) :- goal3, goal4.
                           thisclause(arg4) :- goal5.

==> mv 2 4
      results in          thisclause(arg1).
                           thisclause(arg3) :- goal3, goal4.
                           thisclause(arg4) :- goal5.
                           thisclause(arg2) :- goal1, goal2.
```

EXAMPLE 2:

```
The current structure is  goal1, goal2, goal3, goal4

==> mv 1 3
      results in          goal2, goal3, goal1, goal4
```

- continued -

APPENDIX C

EXAMPLE 3:

The current structure is arg1, arg2, arg3, arg4, arg5

==> mv 2 4
 results in arg1, arg3, arg4, arg2, arg5

EXAMPLE 4:

The current structure is arg1, arg2, arg3, arg4, arg5

==> mv 2 0
 results in arg2, arg1, arg3, arg4, arg5

APPENDIX C

CoPy

`cp <fromarg> <toarg>`

The 'CoPy' command copies either: a single rule, a single goal, or a single item. If the cursor is positioned at the start of a rule, a rule is copied. If the cursor is positioned at a goal in the body of a rule, a goal is copied. If the cursor is positioned at the arguments to a functor, an argument is copied.

The structure at position <fromarg> is copied behind the structure at position <toarg>. If either of the arguments is higher than the number of items present, an error is flagged and the command is ignored. Zero is a valid value for <toarg>.

EXAMPLE 1:

The current structure is `thisclause(arg1).`
 `thisclause(arg2) :- goal1, goal2.`
 `thisclause(arg3) :- goal3, goal4.`
 `thisclause(arg4) :- goal5.`

`==> cp 2 4`
 results in `thisclause(arg1).`
 `thisclause(arg2) :- goal1, goal2.`
 `thisclause(arg3) :- goal3, goal4.`
 `thisclause(arg4) :- goal5.`
 `thisclause(arg2) :- goal1, goal2.`

EXAMPLE 2:

The current structure is `goal1, goal2, goal3, goal4`

`==> cp 1 3`
 results in `goal1, goal2, goal3, goal1, goal4`

- continued -

APPENDIX C

EXAMPLE 3:

The current structure is arg1, arg2, arg3, arg4, arg5

==> cp 2 4
 results in arg1, arg2, arg3, arg4, arg2, arg5

EXAMPLE 4:

The current structure is arg1, arg2, arg3, arg4, arg5

==> cp 2 0
 results in arg2, arg1, arg2, arg3, arg4, arg5

APPENDIX C

CHange

`ch <structure>`

The 'CHange' command replaces the current structure with `<structure>`. There should be no blanks within `<structure>`.

EXAMPLE 1:

The current structure is `goal3, goal4`

```
==> ch goal5,goal6
      results in           goal5, goal6
```

EXAMPLE 2:

The current structure is `goal7(arg1, arg2)`

```
==> ch goal8(arg3,arg4)
      results in           goal8(arg3, arg4)
```

****CAUTION**** This command should be used with great care. No structural checks are made on `<structure>` so invalid structural formats can be created. This command should not be used to add new structures. The commands 'InsertAfter' and 'InsertBefore' ensure proper structural format for additions. When replacing an entire rule, the head and body should be separately modified. The commands 'CHangeArgument', 'CHangeFunctor' and 'CHangeGoal' are recommended for novice users to ensure proper structural format.

APPENDIX C

CHangeArgument

`cha <num> <new>`

The 'CHangeArgument' command replaces the argument in the position of <num> with the value of <new>. There should be no blanks within <new>.

EXAMPLE 1:

The current structure is functor(arg1, arg2, arg3)

==> cha 2 newvalue
 results in functor(arg1, newvalue, arg3)

EXAMPLE 2:

The current structure is goal1(arg1, arg2), goal2

==> cha 2 newvalue
 results in goal1(arg1, arg2), newvalue

NOTE: The structure 'goal1(arg1, arg2), goal2' is stored internally as ',(goal1(arg1, arg2), goal2)'. To modify an argument of goal1, 'Down' must first be executed.

EXAMPLE:

The current structure is goal1(arg1, arg2), goal2

==> d cha 2 newvalue u
 results in goal1(arg1, newvalue), goal2

At this point in the example, goal traversing commands such as 'NextGoal' and 'PreviousGoal' can be executed.

APPENDIX C

CHangeFunctor

`chf <new>`

The 'CHangeFunctor' command replaces the functor in the current structure with the value of <new>. There should be no blanks within <new>.

EXAMPLE:

The current structure is `goal1(arg1, arg2, arg3)`

```
==> chf newfunctor
      results in      newfunctor(arg1, arg1, arg3)
```

APPENDIX C

CHangeGoal

`chg <num> <new>`

The 'CHangeGoal' command replaces the goal in the position of <num> with the value of <new>. There should be no blanks within <new>.

EXAMPLE:

The current structure is goal1,goal2,goal3

```
==> chg 2 newgoal(arg1)
      results in          goal1,newgoal(arg1),goal3
```

This command should not be used to add new goals. The commands 'InsertAfter' and 'InsertBefore' ensure proper structural format for additions.

APPENDIX C

REMove

```
rem <name>
```

The 'REMove' command removes the current structure and places it in the buffer <name>. The buffer <name> should begin in a lower case letter. The resultant structure is the null list. See 'REPLace', 'BUFFER', 'DISPLAY', and 'CONTENTS' for further buffer commands. The buffer will remain in the data base for the rest of the edit session.

APPENDIX C

REPLace

`repl <name>`

The 'REPLace' command replaces the current structure with the contents of buffer <name>. The buffer <name> should begin in a lower case letter. See 'REMove', 'BUFFER', 'DISPLAY', and 'CONTENTS' for further buffer commands.

EXAMPLE:

The current procedure is goal6(arg1).

Buffer 'buff1' contains goal7(arg5)

```
==> repl buff1
      results in             goal7(arg5)
```

****CAUTION**** This command should be used with great care. No structural checks are made on <structure> so invalid structural formats can be created. This command should not be used to add new structures. The commands 'InsertAfter' and 'InsertBefore' ensure proper structural format for additions. When replacing an entire rule, the head and body should be separately modified. The commands 'CHangeArgument', 'CHangeFunctor' and 'CHangeGoal' are recommended for novice users to ensure proper structural format.

APPENDIX C

UNdo

un

The 'UNdo' command attempts to reverse the effects of the last executed command.

If the last command was a traversal command such as 'Up', 'Down', 'Left', or 'Right', the opposite traversal command is executed. Only a single move is made (Example: if last command was 'Down *' or 'Down 5', the 'UNdo' results in the execution of 'Up'). If the last command was 'TOP', 'NextClause' is executed; if the last command was 'Start', 'Head' is executed.

For other commands which can be 'undone', the previous structure replaces the current structure. This occurs for modification commands such as 'CHange', 'InsertAfter', 'DElete', etc.

Commands which can not be 'undone' (ex. 'Find', 'FILE') generate an error message with no effect on the current structure or cursor position.

APPENDIX C

BUFFER

```
buffer <name> <structure>
```

The 'BUFFER' command creates a buffer <name> with contents <structure>. The buffer <name> should begin in a lower case letter. The <structure> should be a PROLOG structure. See 'REMOve', 'REPLace', 'BUFFER' and 'CONTENTS' for further buffer commands.

EXAMPLE:

```
=> buffer buff2 goal2,goal3
    results in a buff2      goal2,goal3
```


APPENDIX C

DISPLAY

display

The 'DISPLAY' command displays the names of all the buffers in the data base. See 'REMove', 'REPLace', 'BUFFER' and 'CONTENTS' for further buffer commands.

APPENDIX C

CONTENTS

`contents <name>`

The 'CONTENTS' command displays the contents of the buffer <name>. The buffer <name> should begin in a lower case letter. See 'REMOve', 'REPLace', 'BUFFER' and 'DISPLAY' for further buffer commands.

APPENDIX C

CHangeDELIMiter

```
chdelim <new>
```

The 'CHangeDELIMiter' command replaces the current delimiter with <new>. The <delim> is equal to '/' (slash) when the edit session begins.

APPENDIX C

RePeaT

rpt

The 'RePeaT' command repeats the last traversal or modification command successfully executed. Only a single repetition is made for traversal commands. For example, if the last command was 'Down *' or 'Down 5', the 'RePeaT' results in the execution of 'Down'.

APPENDIX C

DEFineComManD

```
defcmd <name> <command list> <delim>
```

The 'DEFindeComManD' command defines a new command <name>. This command can be referenced until the end of the edit session and invokes calls to all the commands in the <command list>. The <command list> is composed of existing commands which are entered as defined in the user's manual. The command list is terminated by the current <delim>. The <delim> is equal to '/' (slash) when the edit session begins. (See 'CHangeDELIMiter' to change the current delimiter.) (See 'DISPLComManD' to display the list of all new commands and 'COMMAND' to display the list of associated commands for a new command.)

APPENDIX C

DISPLayComManD

displcmd

The 'DISPlayComManD' command displays all new commands entered within the edit session. (See 'DEFineComManD' to define new commands and 'COMMAND' to display the list of associated commands for a new command.)

APPENDIX C

COMMAND

`command <name>`

The 'COMMAND' command prints the list of commands associated with the new command <name>. (See 'DEFineComMand' to define new commands and 'DISPLComMand' to display the list of all new commands.)

APPENDIX C

Sample Edit Sessions

```
| ?- pedit(head3(A,B)).  
Welcome to PEDIT - a PROLOG-Based Editor  
For assistance, type <help help> .
```

Clause retrieved from data base.

```
PEDIT:t  
head3(x,y) :-  
    rule1,rule2(a,b).  
head3(w,z) :-  
    rule1,rule2,rule3,rule4,rule5.  
head3(_49,_50) :-  
    rule5(rule6(_51,_52),_53),rule7(_49,_50,_54).  
head3(_25,_26).
```

```
PEDIT:nc b ng 3 t  
rule4,rule5
```

```
PEDIT:ia goal1 , goal2 , / t  
rule4,goal1,goal2,rule5
```

```
PEDIT:s nc b ng d r ib A B / 1 * u t  
1 repetition(s)  
rule7(_49,A,B,_50,_54)
```

```
PEDIT:top h cha 2 c t  
head3(x,c)
```

```
PEDIT:s b chg 1 rule23 top t  
head3(x,c) :-  
    rule23,rule2(a,b).  
head3(w,z) :-  
    rule1,rule2,rule3,rule4,goal1,goal2,rule5.  
head3(_49,_50) :-  
    rule5(rule6(_51,_52),_53),rule7(_49,A,B,_50,_54).  
head3(_25,_26).
```

```
PEDIT:mv 1 4 t  
head3(w,z) :-  
    rule1,rule2,rule3,rule4,goal1,goal2,rule5.  
head3(_49,_50) :-  
    rule5(rule6(_51,_52),_53),rule7(_49,A,B,_50,_54).  
head3(_25,_26).  
head3(x,c) :-  
    rule23,rule2(a,b).
```

```
PEDIT:file  
Do you wish to store in a file?  
Enter <yes.> or <no.> yes.  
Type file name: newhead3.  
newhead3 consulted 300 bytes 0.116678 sec.  
Clause replaced in data base
```


APPENDIX C

Source Code

```

/* main routine */
pedit(ClauseName) :-
    init,
    edit([ClauseName], _, _, _, _),
    cleanup.
init :-
    nl,
    write('Welcome to PEDIT - a PROLOG-Based Editor'), nl,
    write(' For assistance, type <help help> .'), nl, nl,
    asserta(delim(/)).
cleanup :-
    retract_all(traversal(_, _)),
    retract_all(cmd_list(_)),
    retract_all(new_command(_, _)),
    retract_all(buffer(_, _)),
    retract_all(last_command(_, _)),
    retract_all(undocommand(_)),
    retract_all(changed(_)),
    retract_all(counter(_)),
    retract_all(delim(_)).
cleanup.

edit([ClauseName | []], _,
    OldClause, OldClause, LastUndo, LastUndo) :-
    get_routine(ClauseName, Routine),
    inform(Routine),
    !,
    remove_old_undo,
    randex(done, ClauseName, Routine, NewRoutine,
        Routine, ThisUndo).

inform(Routine) :-
    not_null(Routine),
    write('Clause retrieved from data base.'), nl.
inform(Routine) :-
    write('Clause not in data base. '), nl,
    write('Clauses may be added via the <ib> command.'), nl.

```

APPENDIX D

```

/* Routine which extracts clauses from the */
/* data base and puts them in internal    */
/* PEDIT format.                          */
get_routine(Name, Routine) :-
    find_all_rules(Name, ListofHeads, ListofTails),
    !, construct_routine(
        ListofHeads, ListofTails, [], Routine).

find_all_rules(Head, _, _) :-
    asserta(rule(mark,mark)),
    clause(Head,Tail),
    asserta(rule(Head,Tail)),
    fail.
find_all_rules(_, H, T) :-
    collect_rules([],[],M,N),
    !, H = M, T = N.
find_all_rules(_,_,_).
collect_rules(S,T,L,M) :-
    get_next_rule(X,Y),
    !, collect_rules([X|S],[Y|T],L,M).
collect_rules(L,M,L,M).
get_next_rule(X,Y) :-
    retract(rule(X,Y)), !, X == mark.
construct_routine([], [], Routine, Routine).
construct_routine([HofHeads | RestofHeads],
    [HofTails | RestofTails],
    InRoutine, TotalRoutine) :-
    ThisRule =.. [if, HofHeads, HofTails],
    app1(InRoutine, [ThisRule], PartRoutine),
    construct_routine(RestofHeads, RestofTails,
        PartRoutine, TotalRoutine).

```

APPENDIX D

```

/* Driving routine for reading and executing commands. */
randex(Until, ClauseName, Structure, NewStructure,
      LastUndo, NewUndo) :-
    get_command([Command | Arglist]),
    Procall =.. [Command, Arglist, ClauseName,
        Structure, TempStructure,
        LastUndo, TempUndo],
    call(Procall),
    continue(Until, Command, ClauseName,
        TempStructure, NewStructure, TempUndo, NewUndo).

continue(Until, Command, ClauseName,
      ThisStructure, ThisStructure, LastUndo, LastUndo) :-
    Until = Command.
continue(Until, Command, ClauseName,
      ThisStructure, NewStructure, LastUndo, NewUndo) :-
    randex(Until, ClauseName, ThisStructure, NewStructure,
        LastUndo, NewUndo).

/* Gets commands from Command List */
/* If none present, reads new line */
/* from user. */
get_command(ReturnedCommand) :-
    retract(cmd_list(ReturnedCommand)).
get_command(ReturnedCommand) :-
    read_new_list,
    get_command(ReturnedCommand).

read_new_list :-
    nl,
    write('PEDIT:'),
    read_in(ListofWords),
    process_list(ListofWords).
read_new_list.
read_in([Word | MoreWords]) :-
    get0(Char),
    read_word(Char, Word, NextChar),
    rest_line(Word, NextChar, MoreWords).
rest_line(Word, Char, []) :-
    newline(Char), !.
rest_line(Word, Char, [Word1 | Words]) :-
    read_word(Char, Word1, NextChar),
    rest_line(Word1, NextChar, Words).
read_word(Char, Word, Char) :-
    newline(Char), !, name(Word, [Char]).
read_word(Char, Word, NextChar) :-
    in_word(Char), !,
    get0(C1),
    rest_word(C1, CharList, NextChar),
    name(Word, [Char | CharList]).
read_word(Char, Word, NextChar) :-
    get0(C1),

```

APPENDIX D

```
    read_word(C1, Word, NextChar).
rest_word(Char, [Char | CharList], NextChar) :-
    in_word(Char), !,
    get0(C1),
    rest_word(C1, CharList, NextChar).
rest_word(Char, [], Char).
in_word(Char) :- Char > 32, Char < 126.
newline(10).
```

APPENDIX D

```

process_list([]).
process_list([EofLine]) :- eofline(EofLine).
process_list([Command | Rest]) :-
    process_command(Command, Rest, NewRest),
    process_list(NewRest).
process_command(quit, Rest, Rest) :-
    assertz(cmd_list([quit])).
process_command(file, Rest, Rest) :-
    assertz(cmd_list([file])).
process_command(end, Rest, Rest) :-
    assertz(cmd_list([end])).
process_command(help, [Name | Rest], Rest) :-
    assertz(cmd_list([help, Name])).
process_command(t, Rest, Rest) :-
    assertz(cmd_list([type])).
process_command(top, Rest, Rest) :-
    assertz(cmd_list([top])).
process_command(h, Rest, Rest) :-
    assertz(cmd_list([head])).
process_command(b, Rest, Rest) :-
    assertz(cmd_list([body])).
process_command(s, Rest, Rest) :-
    assertz(cmd_list([start])).
process_command(nc, [* | Rest], Rest) :-
    assertz(cmd_list([next_clause, *])).
process_command(nc, [Num | Rest], Rest) :-
    integer(Num),
    assertz(cmd_list([next_clause, Num])).
process_command(nc, Rest, Rest) :-
    assertz(cmd_list([next_clause])).
process_command(pc, [* | Rest], Rest) :-
    assertz(cmd_list([previous_clause, *])).
process_command(pc, [Num | Rest], Rest) :-
    integer(Num),
    assertz(cmd_list([previous_clause, Num])).
process_command(pc, Rest, Rest) :-
    assertz(cmd_list([previous_clause])).
process_command(ng, [* | Rest], Rest) :-
    assertz(cmd_list([next_goal, *])).
process_command(ng, [Num | Rest], Rest) :-
    integer(Num),
    assertz(cmd_list([next_goal, Num])).
process_command(ng, Rest, Rest) :-
    assertz(cmd_list([next_goal])).
process_command(pg, [* | Rest], Rest) :-
    assertz(cmd_list([previous_goal, *])).
process_command(pg, [Num | Rest], Rest) :-
    integer(Num),
    assertz(cmd_list([previous_goal, Num])).
process_command(pg, Rest, Rest) :-
    assertz(cmd_list([previous_goal])).

```

APPENDIX D

```

process_command(d, [* | Rest], Rest) :-
    assertz(cmd_list([down, *])).
process_command(d, [Num | Rest], Rest) :-
    integer(Num),
    assertz(cmd_list([down, Num])).
process_command(d, Rest, Rest) :-
    assertz(cmd_list([down])).
process_command(u, [* | Rest], Rest) :-
    assertz(cmd_list([up, *])).
process_command(u, [Num | Rest], Rest) :-
    integer(Num),
    assertz(cmd_list([up, Num])).
process_command(u, Rest, Rest) :-
    assertz(cmd_list([up])).
process_command(r, [* | Rest], Rest) :-
    assertz(cmd_list([right, *])).
process_command(r, [Num | Rest], Rest) :-
    integer(Num),
    assertz(cmd_list([right, Num])).
process_command(r, Rest, Rest) :-
    assertz(cmd_list([right])).
process_command(l, [* | Rest], Rest) :-
    assertz(cmd_list([left, *])).
process_command(l, [Num | Rest], Rest) :-
    integer(Num),
    assertz(cmd_list([left, Num])).
process_command(l, Rest, Rest) :-
    assertz(cmd_list([left])).
process_command(f, [Num | [Pattern | Rest]], Rest) :-
    assertz(cmd_list([find_argument, Num, Pattern])).
process_command(del, [* | Rest], Rest) :-
    assertz(cmd_list([delete_structures, *])).
process_command(del, [Num | Rest], Rest) :-
    integer(Num),
    assertz(cmd_list([delete_structures, Num])).
process_command(del, Rest, Rest) :-
    assertz(cmd_list([delete_structures])).
process_command(ia, [Name | Args], Rest) :-
    find_delim(Args, [], Commands, Rest),
    assertz(cmd_list([insert_after, Name | Commands])).
process_command(ib, [Name | Args], Rest) :-
    find_delim(Args, [], Commands, Rest),
    assertz(cmd_list([insert_before, Name | Commands])).
process_command(mv, [From | [To | Rest]], Rest) :-
    assertz(cmd_list([move_item, From, To])).
process_command(cp, [From | [To | Rest]], Rest) :-
    assertz(cmd_list([copy_item, From, To])).
process_command(ch, [New | Rest], Rest) :-
    assertz(cmd_list([change_structure, New])).
process_command(cha, [Num | [NewVal | Rest]], Rest) :-
    integer(Num),
    assertz(cmd_list([change_argument, Num, NewVal])).

```

APPENDIX D

```

process_command(chf, [New | Rest], Rest) :-
    atom(New),
    assertz(cmd_list([change_this_functor, New])).
process_command(chg, [Num | [NewVal | Rest]], Rest) :-
    integer(Num),
    assertz(cmd_list([change_goal, Num, NewVal])).
process_command(rem, [Name | Rest], Rest) :-
    assertz(cmd_list([remove, Name])).
process_command(repl, [Name | Rest], Rest) :-
    assertz(cmd_list([replace, Name])).
process_command(un, Rest, Rest) :-
    assertz(cmd_list([undo])).
process_command(buffer, [Name | [Contents | Rest]], Rest) :-
    assertz(cmd_list([create_buffer, Name, Contents])).
process_command(display, Rest, Rest) :-
    assertz(cmd_list([display_buffers])).
process_command(contents, [Name | Rest], Rest) :-
    assertz(cmd_list([buffer_contents, Name])).
process_command(chdelim, [New | Rest], Rest) :-
    assertz(cmd_list([change_delimiter, New])).
process_command(rpt, Rest, Rest) :-
    assertz(cmd_list([repeat])).
process_command(defcmd, [Name | Args], Rest) :-
    find_delim(Args, [], Commands, Rest),
    assertz(cmd_list([define_command, Name | Commands])).
process_command(displcmd, Rest, Rest) :-
    assertz(cmd_list([display_commands])).
process_command(command, [Name | Rest], Rest) :-
    assertz(cmd_list([command_contents, Name])).
process_command(32, Rest, Rest).
process_command(Command, Rest, Rest) :-
    new_command(Command, Args),
    add_to_cmd_list(Args).
process_command(Command, Rest, []) :-
    nl,
    write('*****'), nl,
    write('Error in input string starting with '),
    write(Command), nl,
    write('Rest of input string ignored'), nl,
    write('*****'), nl.

find_delim([], Commands, Commands, []) :- !.
find_delim([Head | Tail], Commands, Commands, Tail) :-
    delim(Head), !.
find_delim([Head | Tail], InList, ReturnList, Rest) :-
    app1(InList, [Head], Out1),
    find_delim(Tail, Out1, ReturnList, Rest).

add_to_cmd_list([]) :- !.
add_to_cmd_list([Head | Tail]) :-
    process_command(Head, Tail, NewTail),
    add_to_cmd_list(NewTail).

```

APPENDIX D

```

/* Command procedures */
/* */
/* PARAMETERS: */
/* Arg list - Passed */
/* Clause Name - Passed */
/* Current Structure - Passed */
/* New Structure - Returned */
/* Last Undo - Passed */
/* New Undo - Returned */
/* */
/* These commands are called */
/* by the randex routine. */

/***** quit *****/
quit([], ClauseName, Structure, Structure,
      LastUndo, LastUndo) :-
    changed(ClauseName),
    retract_all(changed(ClauseName)),
    write('Changes have not been saved'),
    nl,
    end([], ClauseName, Structure, Structure,
          LastUndo, LastUndo).
quit([], ClauseName, Structure, Structure,
      LastUndo, LastUndo) :-
    asserta(cmd_list([done])),
    asserta(cmd_list([top])).

/***** file *****/
file([], ClauseName, Structure, Structure,
      LastUndo, LastUndo) :-
    asserta(cmd_list([done])),
    asserta(cmd_list([do_file])),
    asserta(cmd_list([top])).
do_file([], ClauseName, Structure, Structure,
         LastUndo, LastUndo) :-
    retract_and_save(ClauseName),
    retract_all(changed(ClauseName)),
    told,
    file_save(Structure),
    retract_all(saved(X,Y)).
do_file([], ClauseName, Structure, Structure,
         LastUndo, LastUndo) :-
    retract_all(ClauseName),
    restore_clause,
    write('Can not properly format this structure '), nl,
    write('Structure saved in file peditjunk '), nl,
    tell(peditjunk),
    save_structure(Structure),
    told.

```


APPENDIX D

```

done([], ClauseName, Structure, Structure,
    LastUndo, LastUndo).

retract_and_save(X) :-
    clause(X,Y),
    assertz(saved(X,Y)),
    fail.
retract_and_save(X) :- retract_all(X).

restore_clause :-
    retract(saved(X,Y)),
    restore_saved(X,Y),
    fail.
restore_clause.

restore_saved(H,true) :-
    assertz(H).
restore_saved(H,T) :-
    assertz(:-(H,T)).

make_filename(ClaueName, FileName) :-
    ClauseName =.. [Functor | Rest],
    app1([Functor],[t,m,p],FileName).

replace_clause(ClaueName, []).
replace_clause(ClaueName,
    [HeadofStructures | RestofStructures]) :-
    HeadofStructures =.. [if, Head, Tail],
    write_occurrence(Head, Tail),
    write('.'),
    nl,
    replace_clause(ClaueName, RestofStructures).

add_occurrence(Head, true) :-
    assertz(Head).
add_occurrence(Head, Tail) :-
    assertz(:-(Head, Tail)).

file_save(Structure) :-
    write('Do you wish to store in a file? '),
    nl,
    write('Enter <yes.> or <no.> '),
    read(Ans),
    file_response(Ans, Structure).
file_response(yes, Structure) :-
    write('Type file name: '),
    read(Filename),
    tell(Filename),
    save_structure(Structure),
    told,
    consult(Filename),
    write('Clause replaced in data base'), nl.

```

APPENDIX D

```

file_response(y, Structure) :-
    write('Type file name: '),
    read(Filename),
    tell(Filename),
    save_structure(Structure),
    told,
    consult(Filename),
    write('Clause replaced in data base'), nl.
file_response(no, Structure) :-
    tell(pedittmp),
    replace_clause(ClauseName, Structure),
    told,
    consult(pedittmp),
    name(pedittmp, Charlist),
    system([114, 109, 32 | Charlist]),
    write('Clause replaced in data base'), nl.
file_response(n, Structure) :-
    tell(pedittmp),
    replace_clause(ClauseName, Structure),
    told,
    consult(pedittmp),
    name(pedittmp, Charlist),
    system([114, 109, 32 | Charlist]),
    write('Clause replaced in data base'), nl.
file_response(_, Structure) :-
    write('Invalid answer '), nl,
    write('enter <yes.> or <no.> '),
    read(Ans),
    file_response(Ans, Structure).

save_structure([]).
save_structure([HeadofStructures | RestofStructures]) :-
    HeadofStructures =.. [if, Head, Tail],
    write_occurrence(Head, Tail),
    write(' '), nl,
    save_structure(RestofStructures).
save_structure([HeadofStructures | RestofStructures]) :-
    write(HeadofStructures), nl,
    save_structure(RestofStructures).
save_structure(_).
write_occurrence(Head, true) :-
    write(Head).
write_occurrence(Head, Tail) :-
    write((Head :- Tail)).

```

APPENDIX D

```

/***** end *****/
end([], ClauseName, Structure, Structure,
    LastUndo, LastUndo) :-
    write('quit or save? '),
    read(Ans),
    end_response(Ans).
end_response(quit) :-
    asserta(cmd_list([quit])),
    asserta(cmd_list([top])).
end_response(save) :-
    asserta(cmd_list([done])),
    asserta(cmd_list([do_file])),
    asserta(cmd_list([top])).
end_response(_) :-
    write('Invalid answer'),
    nl,
    write('Enter either <quit.> or <save.> '),
    read(Ans),
    end_response(Ans).

/***** help *****/
help([quit], ClauseName, Structure, Structure,
    LastUndo, LastUndo) :-
    write('Format: '), nl,
    write('      quit '), nl, nl,
    write('Aborts the current edit session. '), nl,
    write('If changes have been made but not saved, '), nl,
    write('the user will be prompted for '),
    write('verification. '), nl,
    write('If this occurs, enter either <quit.> '),
    write(' or <save.> '), nl.
help([file], ClauseName, Structure, Structure,
    LastUndo, LastUndo) :-
    write('Format: '), nl,
    write('      file '), nl, nl,
    write('Replaces all occurrences in the data'),
    write(' base '), nl,
    write('with the edited structure. '), nl,
    write('It then asks if the structures are to be '), nl,
    write('stored in a file. '),
    write(' Enter either <yes.> or '), nl,
    write(' <no.>. If <yes.>, enter <filename.>. '), nl.
help([end], ClauseName, Structure, Structure,
    LastUndo, LastUndo) :-
    write('Format: '), nl,
    write('      end '), nl, nl,
    write('Ends the current edit session. '), nl,
    write('Prompts <quit or save?>. '), nl,
    write('Enter either <quit.> or <save.>. '), nl.

```

APPENDIX D

```

help([help], ClauseName, Structure, Structure,
    LastUndo, LastUndo) :-
    write('Format:      '), nl,
    write('      help <command> '), nl, nl,
    write('Commands for which help is available: '), nl,
    write('      General Edit Commands  '), nl, write('      '),
    write('quit      '),
    write('file      '),
    write('end       '),
    write('help      '),
    write('t         '), nl, nl,
    write('      Traversal Commands  '), nl,
    write('      '),
    write('top       '),
    write('nc        '),
    write('pc        '),
    write('h         '),
    write('b         '),
    write('s         '), nl,
    write('      '),
    write('ng        '),
    write('pg        '),
    write('d         '),
    write('u         '),
    write('r         '),
    write('l         '), nl, nl,
    write('      Modification Commands  '), nl,
    write('      '),
    write('f         '),
    write('del       '),
    write('ia        '),
    write('ib        '),
    write('mv        '),
    write('ch        '), nl,
    write('      '),
    write('cha       '),
    write('chf       '),
    write('chg       '),
    write('rem       '),
    write('repl      '), nl, nl,
    write('      Special Editor Commands  '), nl,
    write('      '),
    write('un        '),
    write('buffer    '),
    write('display   '),
    write('contents  '),
    write('chdelim   '),
    write('rpt       '), nl,
    write('      '),
    write('defcmd    '),
    write('displcmd  '),
    write('command   '), nl.

```

APPENDIX D

```

help([t], ClauseName, Structure, Structure,
LastUndo, LastUndo) :-
    write('Format:      '), nl,
    write('      t '), nl, nl,
    write('Causes the current structure to be '), nl,
    write('displayed in pretty printer format. '), nl.
help([top], ClauseName, Structure, Structure,
LastUndo, LastUndo) :-
    write('Format:      '), nl,
    write('      top '), nl, nl,
    write('Moves the cursor to the beginning of the '), nl,
    write('first procedure occurrence. '), nl.
help([nc], ClauseName, Structure, Structure,
LastUndo, LastUndo) :-
    write('Format:      '), nl,
    write('      nc or nc <num> or nc * '), nl, nl,
    write('Moves the cursor to the start of the '), nl,
    write('next procedure occurrence. '), nl,
    write_trav_parms.
help([pc], ClauseName, Structure, Structure,
LastUndo, LastUndo) :-
    write('Format:      '), nl,
    write('      pc or pc <num> or pc * '), nl, nl,
    write('Moves the cursor to the start of the '), nl,
    write('previous procedure occurrence. '), nl,
    write_trav_parms.
help([h], ClauseName, Structure, Structure,
LastUndo, LastUndo) :-
    write('Format:      '), nl,
    write('      h '), nl, nl,
    write('Moves the cursor to the head of the '), nl,
    write('current procedure occurrence. '), nl.
help([b], ClauseName, Structure, Structure,
LastUndo, LastUndo) :-
    write('Format:      '), nl,
    write('      b '), nl, nl,
    write('Moves the cursor to the body of the '), nl,
    write('current procedure occurrence. '), nl.
help([s], ClauseName, Structure, Structure,
LastUndo, LastUndo) :-
    write('Format:      '), nl,
    write('      s '), nl, nl,
    write('Moves the cursor to the start of the '), nl,
    write('current procedure occurrence. '), nl.
help([ng], ClauseName, Structure, Structure,
LastUndo, LastUndo) :-
    write('Format:      '), nl,
    write('      ng or ng <num> or ng * '), nl, nl,
    write('Moves the cursor to the next goal in the '), nl,
    write('body of the current procedure '),
    write('occurrence. '), nl,
    write_trav_parms.

```

APPENDIX D

```

help([pg], ClauseName, Structure, Structure,
     LastUndo, LastUndo) :-
    write('Format:      '), nl,
    write('      pg or pg <num> or pg * '), nl, nl,
    write('Moves the cursor to the previous '),
    write('goal in the body'), nl,
    write('of the current procedure occurrence. '), nl,
    write_trav_parms.
help([d], ClauseName, Structure, Structure,
     LastUndo, LastUndo) :-
    write('Format:      '), nl,
    write('      d or d <num> or d * '), nl, nl,
    write('Moves the cursor to a lower '),
    write('structure level. '), nl,
    write_trav_parms.
help([u], ClauseName, Structure, Structure,
     LastUndo, LastUndo) :-
    write('Format:      '), nl,
    write('      u or u <num> or u * '), nl, nl,
    write('Moves the cursor to a higher '),
    write('structure level. '), nl,
    write_trav_parms.
help([r], ClauseName, Structure, Structure,
     LastUndo, LastUndo) :-
    write('Format:      '), nl,
    write('      r or r <num> or r * '), nl, nl,
    write('Moves the cursor right in the '),
    write('structure. '), nl,
    write('NOTE: Due to the internal storage '),
    write('mechanism, '), nl,
    write('<r> should not be used to travel '),
    write('among goals. '), nl,
    write('Instead, the command <ng> is provided. '), nl,
    write_trav_parms.
help([l], ClauseName, Structure, Structure,
     LastUndo, LastUndo) :-
    write('Format:      '), nl,
    write('      l or l <num> or l * '), nl, nl,
    write('Moves the cursor left in the structure. '), nl,
    write_trav_parms.
help([f], ClauseName, Structure, Structure,
     LastUndo, LastUndo) :-
    write('Format:      '), nl,
    write('      f <num> <pattern> '), nl, nl,
    write('Searches the bodies of the rules for a '), nl,
    write('match of <pattern> in the '),
    write('argument <num>. '), nl,
    write('If no match is found, the cursor '),
    write('is positioned '), nl,
    write('at the last rule. If a match is found, '),
    write('the cursor '), nl,
    write('is positioned at that rule. '), nl.

```

APPENDIX D

```

help([del], ClauseName, Structure, Structure,
LastUndo, LastUndo) :-
    write('Format:      '), nl,
    write('      del or del <num> or del * '), nl, nl,
    write('Deletes either: single/multiple rules, '),
    write('single/multiple '), nl,
    write('goals, or single/multiple items. '),
    write('If there '), nl,
    write('are fewer items in the current structure '),
    write('than are '), nl,
    write('specified by <num>, all the items '),
    write('are deleted. '), nl,
    write('When no argument is entered, only 1 item '),
    write('is deleted. '), nl, nl,
    write('When <num> is entered, that number of '),
    write('items are deleted. '), nl, nl,
    write('When * is entered, all the items '),
    write('are deleted. '), nl, nl,
    write('If the cursor is positioned at the '),
    write('start of a rule, '), nl,
    write('rules are deleted. If the cursor is '),
    write('positioned at a '), nl,
    write('goal, goals are deleted. Otherwise, '),
    write('items are deleted. '), nl.
help([ia], ClauseName, Structure, Structure,
LastUndo, LastUndo) :-
    write('Format:      '), nl,
    write('      ia <new items> <delim> '), nl, nl,
    write('Inserts either a single rule, '),
    write('single/multiple goals or '), nl,
    write('or single/multiple items after the '),
    write('current structure. '), nl,
    write('If the cursor is positioned at the '),
    write('start of a rule, '), nl,
    write('a new rule is inserted. If the cursor '),
    write('is positioned at a '), nl,
    write('goal, goal(s) are inserted. (Goals '),
    write('should be followed by a goal '), nl,
    write('functor - a comma or semicolon.) Otherwise, '),
    write('item(s) are '), nl,
    write('inserted. The <new items> should be '),
    write('separated by blanks. '), nl,
    write('The <delim> is equal to </> (slash) '),
    write('upon entering PEDIT. '), nl.
help([ib], ClauseName, Structure, Structure,
LastUndo, LastUndo) :-
    write('Format:      '), nl,
    write('      ib <new items> <delim> '), nl, nl,
    write('Inserts either a single rule, '),
    write('single/multiple goals or '), nl,
    write('or single/multiple items before the '),
    write('current structure. '), nl,

```

APPENDIX D

```

write('If the cursor is positioned at the '),
write('start of a rule, '), nl,
write('a new rule is inserted. If the cursor is '),
write('positioned at a '), nl,
write('goal, goal(s) are inserted. (Goals '),
write('should be followed by a goal '), nl,
write('functor - a comma or semicolon.) Otherwise, '),
write('item(s) are '), nl,
write('inserted. The <new items> should be '),
write('separated by blanks. '), nl,
write('The <delim> is equal to </> (slash) '),
write('upon entering PEDIT. '), nl.
help([mv], ClauseName, Structure, Structure,
LastUndo, LastUndo) :-
write('Format: '), nl,
write('      mv <fromarg> <toarg> '), nl, nl,
write('Moves either a single rule, a single '),
write('goal or '), nl,
write('or a single item. If the cursor is '),
write('positioned '), nl,
write('at the start of a rule, a rule is moved. '),
write('If '), nl,
write('the cursor is positioned at a goal, '),
write('a goal is moved. '), nl,
write('Otherwise, an item is moved. '),
write('The structure at position '), nl,
write('<fromarg> is moved behind the structure at '),
write('position <toarg> '), nl.
help([cp], ClauseName, Structure, Structure,
LastUndo, LastUndo) :-
write('Format: '), nl,
write('      cp <fromarg> <toarg> '), nl, nl,
write('Copies either a single rule, a single '),
write('goal or '), nl,
write('or a single item. If the cursor is '),
write('positioned '), nl,
write('at the start of a rule, a rule is copied. '),
write('If '), nl,
write('the cursor is positioned at a goal, '),
write('a goal is moved. '), nl,
write('Otherwise, an item is copied. '),
write('The structure at position '), nl,
write('<fromarg> is copied behind the structure at '),
write('position <toarg> '), nl.
help([ch], ClauseName, Structure, Structure,
LastUndo, LastUndo) :-
write('Format: '), nl,
write('      ch <new> '), nl, nl,
write('Replaces the current structure '), nl,
write('with the value of <new> '), nl,
write('There should be no blanks within <new> '), nl,
write('***CAUTION*** '), nl,

```


APPENDIX D

```

write('This command should be used with '),
write('extreme caution. '), nl,
write('Commands <cha>, <chf> and '),
write('<chg> are '), nl,
write('recommended for novice users to '),
write('ensure proper '), nl,
write('structural format. '), nl.
help([cha], ClauseName, Structure, Structure,
LastUndo, LastUndo) :-
write('Format: '), nl,
write('      cha <num> <new> '), nl, nl,
write('Replaces the argument in the position '), nl,
write('of <num> with the value of <new>. '), nl,
write('There should be no blanks within <new>. '), nl.
help([chf], ClauseName, Structure, Structure,
LastUndo, LastUndo) :-
write('Format: '), nl,
write('      chf <new> '), nl, nl,
write('Replaces the functor in the '),
write('current structure '), nl,
write('with the value of <new>. '), nl,
write('There should be no blanks within <new>. '), nl.
help([chg], ClauseName, Structure, Structure,
LastUndo, LastUndo) :-
write('Format: '), nl,
write('      chg <num> <new> '), nl, nl,
write('Replaces the goal in the position '), nl,
write('of <num> within the body with the '),
write('value of <new>. '), nl,
write('There should be no blanks within <new>. '), nl.
help([rem], ClauseName, Structure, Structure,
LastUndo, LastUndo) :-
write('Format: '), nl,
write('      rem <name> '), nl, nl,
write('Removes the current structure and '),
write('places it in '), nl,
write('in the buffer <name>. The buffer '),
write('<name> should '), nl,
write('begin in a lower case letter. '),
write('The resultant '), nl,
write('structure will be the null list. '), nl.
help([repl], ClauseName, Structure, Structure,
LastUndo, LastUndo) :-
write('Format: '), nl,
write('      repl <name> '), nl, nl,
write('Replaces the current structure with '),
write('the contents '), nl,
write('of the buffer <name>. The buffer '),
write('<name> should '), nl,
write('begin in a lower case letter. '), nl.

```

APPENDIX D

```

help([un], ClauseName, Structure, Structure,
    LastUndo, LastUndo) :-
    write('Format: '), nl,
    write('      un      '), nl, nl,
    write('Reverses the effects of the last '),
    write('executed command. '), nl,
    write('Commands such as <t>, <file> '),
    write('and <contents> '), nl,
    write('can not be undone. '), nl.
help([buffer], ClauseName, Structure, Structure,
    LastUndo, LastUndo) :-
    write('Format: '), nl,
    write('      buffer <name> '), nl, nl,
    write('Creates a buffer <name> with '),
    write('contents <structure>. '), nl,
    write('The <structure> should be a '),
    write('PROLOG structure. '), nl,
    write('The buffer <name> should begin in a '),
    write('lower case letter. '), nl.
help([display], ClauseName, Structure, Structure,
    LastUndo, LastUndo) :-
    write('Format: '), nl,
    write('      display '), nl, nl,
    write('Displays the names of all the buffers '),
    write('in the data base. '), nl.
help([contents], ClauseName, Structure, Structure,
    LastUndo, LastUndo) :-
    write('Format: '), nl,
    write('      contents <name> '), nl, nl,
    write('Displays the contents of the '),
    write('buffer <name>. '), nl,
    write('The buffer <name> should begin in a '),
    write('lower case letter. '), nl.
help([chdelim], ClauseName, Structure, Structure,
    LastUndo, LastUndo) :-
    write('Format: '), nl,
    write('      chdelim <new> '), nl, nl,
    write('Replaces the current delimiter '),
    write('with <new>. '), nl,
    write('The <delim> is equal to </> (slash) '), nl,
    write('upon entering PEDIT. '), nl.
help([rpt], ClauseName, Structure, Structure,
    LastUndo, LastUndo) :-
    write('Format: '), nl,
    write('      rpt      '), nl, nl,
    write('Repeats the last executed traversal '),
    write('or modification '), nl,
    write('command. '), nl.
help([defcmd], ClauseName, Structure, Structure,
    LastUndo, LastUndo) :-
    write('Format: '), nl,
    write('      defcmd <name> <command list> '),

```

APPENDIX D

```

write(' delim> '), nl, nl,
write('Defines a new command <name> '),
write('which can be '), nl,
write('referenced until the end of the '),
write('edit session. '), nl,
write('The new command <name> should begin '),
write('in a lower case letter. '), nl,
write('The new command will invoke calls '),
write('to all the '), nl,
write('commands in the <command list>. The '), nl,
write('<command list> is composed of '),
write('existing commands '), nl,
write('which are entered as defined in the '),
write('user"s manual. '), nl,
write('The <command list> is terminated by the '),
write('current <delim>. '), nl,
write('The <delim> is equal to </> (slash) '),
write('upon entering PEDIT. '), nl.
help([displcmd], ClauseName, Structure, Structure,
LastUndo, LastUndo) :-
write('Format:      '), nl,
write('      displcmd '), nl, nl,
write('Displays the names of all new commands '),
write('entered within '), nl,
write('the edit session. '), nl.
help([command], ClauseName, Structure, Structure,
LastUndo, LastUndo) :-
write('Format:      '), nl,
write('      command <name> '), nl, nl,
write('Prints the list of commands '),
write('associated with '), nl,
write(' the new command <name>. '), nl,
write('The command <name> should begin in a '),
write('lower case letter. '), nl.
help([Arg | []], ClauseName, Structure, Structure,
LastUndo, LastUndo) :-
write('No help available for '),
write(Arg), nl.

write_trav_parms :-
nl,
write('When no argument is entered, '), nl,
write(' only 1 move is made. '), nl, nl,
write('When <num> is entered, '), nl,
write(' that number of moves are made. '), nl, nl,
write('When * is entered, '), nl,
write(' the maximum allowable moves are made. '), nl.

```

APPENDIX D

```

/***** t *****/
type([], ClauseName, Structure, Structure,
    LastUndo, LastUndo) :-
    !, pp(Structure), nl.
pp([]) :- nl.
pp([Front | End]) :-
    Front =.. [if | [Head | Body]],
    Body = [true],
    write(Head), write('.'), nl,
    pp(End).
pp([Front | End]) :-
    Front =.. [if | [Head | Body]],
    write(Head),
    write(' :'), write('-'), nl,
    write(' '), pp(Body), write('.'), nl,
    pp(End).
pp([Front | []]) :-
    write(Front).
pp([Front | End]) :-
    Front =.. [Functor | [Head | Body]],
    comma_or_semicolon(Functor),
    write(Front), write(' '),
    pp(End).
pp([Front | End]) :-
    write(Front), write(' '),
    pp(End).
pp(_).

/***** top *****/
top([], ClauseName, Structure, Structure,
    LastUndo, LastUndo) :-
    build_top_cmd_list(ClauseName),
    add_new_to_old_cmd_list(ClauseName),
    leave_trail(top, [], ClauseName).
build_top_cmd_list(ClauseName) :-
    retract(traversal(ClauseName, Command)),
    traversal_command(Command, _, OppCommand),
    asserta(old_traversal(Command)),
    asserta(new_cmd_list([OppCommand])),
    fail.
build_top_cmd_list(ClauseName).
add_new_to_old_cmd_list(ClauseName) :-
    retract(old_traversal(Name1)),
    asserta(traversal(ClauseName, Name1)),
    retract(new_cmd_list(Name2)),
    asserta(cmd_list(Name2)),
    fail.
add_new_to_old_cmd_list(ClauseName).

```

APPENDIX D

```

/***** nc *****/
next_clause([], ClauseName, [H | Tail] , [H | NewTail],
    LastUndo, NewUndo) :-
    H =.. [ if | _ ],
    not_null(Tail),
    leave_trail(next_clause, [], ClauseName),
    randex(previous_clause, ClauseName, Tail, NewTail,
        LastUndo, NewUndo).
next_clause([*], ClauseName, [H | Tail] , [H | NewTail],
    LastUndo, NewUndo) :-
    H =.. [ if | _ ],
    not_null(Tail),
    leave_trail(next_clause, [*], ClauseName),
    asserta(counter(next_clause)),
    randex(previous_clause, ClauseName, Tail, NewTail,
        LastUndo, NewUndo).
next_clause([*], ClauseName, Structure, Structure,
    LastUndo, LastUndo) :-
    count(next_clause, Structure).
next_clause([0], ClauseName, Structure, Structure,
    LastUndo, LastUndo) :-
    retract_all(counter(next_clause)).
next_clause([Num], ClauseName, [H | Tail] , [H | NewTail],
    LastUndo, NewUndo) :-
    integer(Num),
    H =.. [ if | _ ],
    not_null(Tail),
    leave_trail(next_clause, [Num], ClauseName),
    asserta(counter(next_clause)),
    randex(previous_clause, ClauseName, Tail, NewTail,
        LastUndo, NewUndo).
next_clause([Num], ClauseName, Structure, Structure,
    LastUndo, LastUndo) :-
    integer(Num),
    count(next_clause, Structure).
next_clause(_, ClauseName, Structure, Structure,
    LastUndo, LastUndo) :-
    pedit_error(next_clause, Structure).

/***** pc *****/
previous_clause([], ClauseName, Structure, Structure,
    LastUndo, LastUndo) :-
    last_traversal(next_clause, ClauseName),
    leave_trail(previous_clause, [], ClauseName),
    remove_traversal(next_clause, ClauseName).
previous_clause([*], ClauseName, Structure, Structure,
    LastUndo, LastUndo) :-
    last_traversal(next_clause, ClauseName),
    leave_trail(previous_clause, [*], ClauseName),
    asserta(counter(previous_clause)),
    remove_traversal(next_clause, ClauseName).

```

APPENDIX D

```

previous_clause([*], ClauseName, Structure, Structure,
    LastUndo, LastUndo) :-
    count(previous_clause, Structure).
previous_clause([0], ClauseName, Structure, Structure,
    LastUndo, LastUndo) :-
    retract_all(counter(previous_clause)).
previous_clause([Num], ClauseName, Structure, Structure,
    LastUndo, LastUndo) :-
    last_traversal(next_clause, ClauseName),
    integer(Num),
    leave_trail(previous_clause, [Num], ClauseName),
    asserta(counter(previous_clause)),
    remove_traversal(next_clause, ClauseName).
previous_clause([Num], ClauseName, Structure, Structure,
    LastUndo, LastUndo) :-
    integer(Num),
    count(previous_clause, Structure).
previous_clause(_, ClauseName, Structure, Structure,
    LastUndo, LastUndo) :-
    pedit_error(previous_clause, Structure).

/***** h *****/
head(_, ClauseName,
    [CurrentStructure | Rest], [NewStructure | Rest],
    LastUndo, NewUndo) :-
    CurrentStructure =.. [if | [Head | T]],
    leave_trail(head, [], ClauseName),
    randex(start, ClauseName, [Head], [NewHead | _],
        LastUndo, NewUndo),
    NewStructure =.. [if | [NewHead | T]].
head(_, ClauseName, Structure, Structure,
    LastUndo, LastUndo) :-
    pedit_error(head, Structure).

/***** b *****/
body(_, ClauseName,
    [CurrentStructure | Rest], [NewStructure | Rest],
    LastUndo, NewUndo) :-
    CurrentStructure =.. [if | [H | Tail]],
    leave_trail(body, [], ClauseName),
    randex(start, ClauseName, Tail, NewTail,
        LastUndo, NewUndo),
    NewStructure =.. [if | [H | NewTail]].
body(_, ClauseName, Structure, Structure,
    LastUndo, LastUndo) :-
    pedit_error(body, Structure).

```

APPENDIX D

```

/***** s *****/
start(_, ClauseName, Structure, Structure,
      LastUndo, LastUndo) :-
    last_traversal(body, ClauseName),
    leave_trail(start, [], ClauseName),
    remove_traversal(body, ClauseName).
start(_, ClauseName, Structure, Structure,
      LastUndo, LastUndo) :-
    last_traversal(head, ClauseName),
    leave_trail(start, [], ClauseName),
    remove_traversal(head, ClauseName).
start(_, ClauseName, Structure, Structure,
      LastUndo, LastUndo) :-
    build_start_cmd_list,
    add_new_to_old_cmd_list(ClauseName),
    leave_trail(start, [], ClauseName).
build_start_cmd_list :-
    retract(traversal(ClauseName, Command)),
    asserta(old_traversal(Command)),
    not_next_clause(Command),
    traversal_command(Command, _, OppCommand),
    asserta(new_cmd_list([OppCommand])),
    fail.
build_start_cmd_list.
not_next_clause(next_clause) :- !, fail.
not_next_clause(_).

/***** ng *****/
next_goal([_], ClauseName, [true], [true],
          LastUndo, LastUndo) :-
    count(next_goal, [true]).
next_goal([], ClauseName, [Head | T], [NewHead | T],
          LastUndo, NewUndo) :-
    Head =.. [ Functor | [HeadArg | TailArg]],
    comma_or_semicolon(Functor),
    leave_trail(next_goal, [], ClauseName),
    randex(previous_goal, ClauseName, TailArg, NewTailArg,
            LastUndo, NewUndo),
    create_new_goals(Functor, HeadArg, NewTailArg, NewHead).
next_goal([*], ClauseName, [Head | T], [NewHead | T],
          LastUndo, NewUndo) :-
    Head =.. [ Functor | [HeadArg | TailArg]],
    comma_or_semicolon(Functor),
    leave_trail(next_goal, [*], ClauseName),
    asserta(counter(next_goal)),
    randex(previous_goal, ClauseName, TailArg, NewTailArg,
            LastUndo, NewUndo),
    create_new_goals(Functor, HeadArg, NewTailArg, NewHead).
next_goal([*], ClauseName, [H | T], [H | T],
          LastUndo, NewUndo) :-
    count(next_goal, [H | T]).

```


APPENDIX D

```

next_goal([0], ClauseName, Structure, Structure,
    LastUndo, LastUndo) :-
    retract_all(counter(next_goal)).
next_goal([Num], ClauseName, [Head | T], [NewHead | T],
    LastUndo, NewUndo) :-
    Head =.. [ Functor | [HeadArg | TailArg]],
    comma_or_semicolon(Functor),
    integer(Num),
    leave_trail(next_goal, [Num], ClauseName),
    asserta(counter(next_goal)),
    randex(previous_goal, ClauseName, TailArg, NewTailArg,
        LastUndo, NewUndo),
    create_new_goals(Functor, HeadArg, NewTailArg, NewHead).
next_goal([Num], ClauseName, Structure, Structure,
    LastUndo, LastUndo) :-
    integer(Num),
    count(next_goal, Structure).
next_goal(_, ClauseName, Structure, Structure,
    LastUndo, LastUndo) :-
    peditt_error(next_goal, Structure).

create_new_goals(Fn, HArg, [], HArg).
create_new_goals(Fn, HArg, TArg, NewGoals) :-
    NewGoals =.. [Fn | [HArg | TArg]].

/***** pg *****/
previous_goal([], ClauseName, Structure, Structure,
    LastUndo, LastUndo) :-
    last_traversal(next_goal, ClauseName),
    leave_trail(previous_goal, [], ClauseName),
    remove_traversal(next_goal, ClauseName).
previous_goal([*], ClauseName, Structure, Structure,
    LastUndo, LastUndo) :-
    last_traversal(next_goal, ClauseName),
    leave_trail(previous_goal, [*], ClauseName),
    asserta(counter(previous_goal)),
    remove_traversal(next_goal, ClauseName).
previous_goal([*], ClauseName, Structure, Structure,
    LastUndo, LastUndo) :-
    count(previous_goal, Structure).
previous_goal([0], ClauseName, Structure, Structure,
    LastUndo, LastUndo) :-
    retract_all(counter(previous_goal)).
previous_goal([Num], ClauseName, Structure, Structure,
    LastUndo, LastUndo) :-
    last_traversal(next_goal, ClauseName),
    integer(Num),
    leave_trail(previous_goal, [Num], ClauseName),
    asserta(counter(previous_goal)),
    remove_traversal(next_goal, ClauseName).

```


APPENDIX D

```

previous_goal([Num], ClauseName, Structure, Structure,
    LastUndo, LastUndo) :-
    integer(Num),
    count(previous_goal, Structure).
previous_goal(_, ClauseName, Structure, Structure,
    LastUndo, LastUndo) :-
    pedited_error(previous_goal, Structure).

/***** d *****/
down([*], ClauseName,
    [CurrentStructure | Rest], [CurrentStructure | Rest],
    LastUndo, NewUndo) :-
    CurrentStructure =.. [F | []],
    count(down, CurrentStructure).
down([Num], ClauseName,
    [CurrentStructure | Rest], [CurrentStructure | Rest],
    LastUndo, NewUndo) :-
    integer(Num),
    CurrentStructure =.. [F | []],
    count(down, CurrentStructure).
down(_, ClauseName,
    [CurrentStructure | Rest], [CurrentStructure | Rest],
    LastUndo, NewUndo) :-
    CurrentStructure =.. [F | []],
    pedited_error(down, CurrentStructure).
down([], ClauseName,
    [CurrentStructure | Rest], [NewStructure | Rest],
    LastUndo, NewUndo) :-
    CurrentStructure =.. [Functor | Arglist],
    leave_trail(down, [], ClauseName),
    randex(up, ClauseName, Arglist, NewArglist,
        LastUndo, NewUndo),
    NewStructure =.. [Functor | NewArglist].
down([*], ClauseName,
    [CurrentStructure | Rest], [NewStructure | Rest],
    LastUndo, NewUndo) :-
    CurrentStructure =.. [Functor | Arglist],
    leave_trail(down, [*], ClauseName),
    assert(counter(down)),
    randex(up, ClauseName, Arglist, NewArglist,
        LastUndo, NewUndo),
    NewStructure =.. [Functor | NewArglist].
down([*], ClauseName, Structure, Structure,
    LastUndo, LastUndo) :-
    count(down, Structure).
down([0], ClauseName, Structure, Structure,
    LastUndo, LastUndo) :-
    retract_all(counter(down)).

```

APPENDIX D

```

down([Num], ClauseName,
     [CurrentStructure | Rest], [NewStructure | Rest],
     LastUndo, NewUndo) :-
    CurrentStructure =.. [Functor | Arglist],
    integer(Num),
    leave_trail(down, [Num], ClauseName),
    asserta(counter(down)),
    randex(up, ClauseName, Arglist, NewArglist,
           LastUndo, NewUndo),
    NewStructure =.. [Functor | NewArglist].
down([Num], ClauseName, Structure, Structure,
     LastUndo, LastUndo) :-
    integer(Num),
    count(down, Structure).
down(_, ClauseName, Structure, Structure,
     LastUndo, LastUndo) :-
    pedir_error(down, Structure).

/***** u *****/
up([], ClauseName, Structure, Structure,
   LastUndo, LastUndo) :-
    last_traversal(down, ClauseName),
    leave_trail(up, [], ClauseName),
    remove_traversal(down, ClauseName).
up([*], ClauseName, Structure, Structure,
   LastUndo, LastUndo) :-
    last_traversal(down, ClauseName),
    leave_trail(up, [*], ClauseName),
    asserta(counter(up)),
    remove_traversal(down, ClauseName).
up([*], ClauseName, Structure, Structure,
   LastUndo, LastUndo) :-
    count(up, Structure).
up([0], ClauseName, Structure, Structure,
   LastUndo, LastUndo) :-
    retract_all(counter(up)).
up([Num], ClauseName, Structure, Structure,
   LastUndo, LastUndo) :-
    last_traversal(down, ClauseName),
    integer(Num),
    leave_trail(up, [Num], ClauseName),
    asserta(counter(up)),
    remove_traversal(down, ClauseName).
up([Num], ClauseName, Structure, Structure,
   LastUndo, LastUndo) :-
    integer(Num),
    count(up, Structure).
up(_, ClauseName, Structure, Structure,
   LastUndo, LastUndo) :-
    pedir_error(up, Structure).

```

APPENDIX D

```

/***** r *****/
right([*], ClauseName, [H | []], [H],
      LastUndo, NewUndo) :-
    count(right, [H]).
right([Num], ClauseName, [H | []], [H],
      LastUndo, NewUndo) :-
    integer(Num),
    count(right, [H]).
right(_, ClauseName, [H | []], [H],
      LastUndo, NewUndo) :-
    pedir_error(right, [H]).
right([], ClauseName, [H | Tail], [H | NewTail],
      LastUndo, NewUndo) :-
    leave_trail(right, [], ClauseName),
    randex(left, ClauseName, Tail, NewTail,
           LastUndo, NewUndo).
right([*], ClauseName, [H | Tail], [H | NewTail],
      LastUndo, NewUndo) :-
    leave_trail(right, [*], ClauseName),
    asserta(counter(right)),
    randex(left, ClauseName, Tail, NewTail,
           LastUndo, NewUndo).
right([0], ClauseName, Structure, Structure,
      LastUndo, LastUndo) :-
    retract_all(counter(right)).
right([Num], ClauseName, [H | Tail], [H | NewTail],
      LastUndo, NewUndo) :-
    integer(Num),
    leave_trail(right, [Num], ClauseName),
    asserta(counter(right)),
    randex(left, ClauseName, Tail, NewTail,
           LastUndo, NewUndo).
right(_, ClauseName, Structure, Structure,
      LastUndo, LastUndo) :-
    pedir_error(right, Structure).

/***** l *****/
left([], ClauseName, Tail, Tail,
      LastUndo, LastUndo) :-
    last_traversal(right, ClauseName),
    leave_trail(left, [], ClauseName),
    remove_traversal(right, ClauseName).
left([*], ClauseName, Tail, Tail,
      LastUndo, LastUndo) :-
    last_traversal(right, ClauseName),
    leave_trail(left, [*], ClauseName),
    asserta(counter(left)),
    remove_traversal(right, ClauseName).
left([0], ClauseName, Structure, Structure,
      LastUndo, LastUndo) :-

```

APPENDIX D

```

    retract_all(counter(left)).
left([Num], ClauseName, Tail, Tail,
    LastUndo, LastUndo) :-
    last_traversal(right, ClauseName),
    integer(Num),
    leave_trail(left, [Num], ClauseName),
    asserta(counter(left)),
    remove_traversal(right, ClauseName).
left([*], ClauseName, Tail, Tail,
    LastUndo, LastUndo) :-
    count(left, Tail).
left([Num], ClauseName, Tail, Tail,
    LastUndo, LastUndo) :-
    integer(Num),
    count(left, Tail).
left(_, ClauseName, Tail, Tail,
    LastUndo, LastUndo) :-
    pedited_error(left, Tail).

/***** f *****/
find_argument(Args, ClauseName,
    [Structure | Rest], [Structure | Rest],
    LastUndo, LastUndo) :-
    Structure =.. [Functor | RestArgs],
    not_if(Functor),
    asserta(cmd_list([find_argument | Args])),
    asserta(cmd_list([start])).
find_argument([Num, Pattern], ClauseName,
    [Clause | Rest], [Clause | Rest],
    LastUndo, LastUndo) :-
    Clause =.. [if | [Head | Tail]],
    arg(Num, Head, Value),
    Value == Pattern,
    leave_trail(find_argument, [Num, Pattern], ClauseName).
find_argument(Args, ClauseName, [Clause | []], [Clause],
    LastUndo, LastUndo) :-
    write(' ***rule not found*** '), nl, nl.
find_argument(Args, ClauseName, Structure, Structure,
    LastUndo, LastUndo) :-
    asserta(cmd_list([find_argument | Args])),
    asserta(cmd_list([next_clause])).
find_argument(Args, ClauseName, Structure, Structure,
    LastUndo, LastUndo) :-
    pedited_error(find_argument, Structure).

```

APPENDIX D

```

/***** del *****/
delete_structures([*], ClauseName, Structure, [],
    LastUndo, Structure) :-
    leave_trail(delete_structures, [*], ClauseName).
delete_structures([], ClauseName, Structure, NewStructure,
    LastUndo, Structure) :-
    delete_items(1, Structure, NewStructure),
    leave_trail(delete_structures, [], ClauseName).
delete_structures([Num], ClauseName,
    Structure, NewStructure,
    LastUndo, Structure) :-
    delete_items(Num, Structure, NewStructure),
    leave_trail(delete_structures, [Num], ClauseName).
delete_structures(Args, ClauseName, Structure, Structure,
    LastUndo, LastUndo) :-
    pediat_error(delete_structures, Structure).

delete_items(_, [], []).
delete_items(0, List, List).
delete_items(Num, [Head], NewHead) :-
    Head =.. [Functor | [Goal1 | RestGoals]],
    comma_or_semicolon(Functor),
    NewNum is Num - 1,
    delete_items(NewNum, RestGoals, NewHead).
delete_items(Num, [Head | Tail], NewList) :-
    NewNum is Num - 1,
    delete_items(NewNum, Tail, NewList).

/***** ia *****/
insert_after([NewHead | []], ClauseName,
    [Head | Tail], [Head | [NewStructure | Tail]],
    LastUndo, [Head | Tail]) :-
    Head =.. [if | Rest],
    NewStructure =.. [if, NewHead, true],
    leave_trail(insert_after,
        [c | [NewHead | []]], ClauseName).
insert_after([NewHead | NewGoals], ClauseName,
    [Head | Tail], [Head | [NewStructure | Tail]],
    LastUndo, [Head | Tail]) :-
    Head =.. [if | Rest],
    ib_goals(NewGoals, [], NewTail),
    NewStructure =.. [if, NewHead, NewTail],
    leave_trail(insert_after,
        [c | [NewHead | NewGoals]], ClauseName).
insert_after(Goals, ClauseName, [Head | T], [NewHead | T],
    LastUndo, [Head | T]) :-
    Head =.. [Functor | [Goal1 | [RestGoals | []]]],
    comma_or_semicolon(Functor),
    ib_goals(Goals, RestGoals, NewGoals),
    goal_functor(F),
    NewHead =.. [F, Goal1, NewGoals],

```

APPENDIX D

```

    leave_trail(insert_after, [g | Goals], ClauseName).
insert_after(Goals, ClauseName, [Head | T], [NewHead | T],
    LastUndo, [Head | T]) :-
    last_traversal(next_goal, ClauseName),
    ib_goals(Goals, [], NewGoals),
    goal_functor(F),
    NewHead =.. [F, Head, NewGoals],
    leave_trail(insert_after, [g | Goals], ClauseName).
insert_after(Goals, ClauseName, [Head | T], [NewHead | T],
    LastUndo, [Head | T]) :-
    last_traversal(body, ClauseName),
    ib_goals(Goals, [], NewGoals),
    goal_functor(F),
    NewHead =.. [F, Head, NewGoals],
    leave_trail(insert_after, [g | NewGoals], ClauseName).
insert_after(NewItems, ClauseName, [Head | Tail], [Head | Tail],
    LastUndo, LastUndo) :-
    Head =.. [if | _],
    pedited_error(ia, Structure).
insert_after(NewItems, ClauseName, [Head | Tail], [Head | Tail],
    LastUndo, LastUndo) :-
    Head =.. [Functor | _],
    comma_or_semicolon(Functor),
    pedited_error(ia, Structure).
insert_after(NewArgs, ClauseName,
    [HofList | RestofList], ReturnedList,
    LastUndo, [HofList | RestofList]) :-
    app1([HofList], NewArgs, NewList),
    app1(NewList, RestofList, ReturnedList),
    leave_trail(insert_after, NewItems, ClauseName).
insert_after(NewItems, ClauseName, Structure, Structure,
    LastUndo, LastUndo) :-
    pedited_error(ia, Structure).

/***** ib *****/
insert_before([NewHead | []], ClauseName,
    [Head | Tail], [NewStructure | [Head | Tail]],
    LastUndo, [Head | Tail]) :-
    Head =.. [if | Rest],
    NewStructure =.. [if, NewHead, true],
    leave_trail(insert_before,
        [NewHead | []], ClauseName).
insert_before([NewHead | NewGoals], ClauseName,
    [Head | Tail], [NewStructure | [Head | Tail]],
    LastUndo, [Head | Tail]) :-
    Head =.. [if | Rest],
    ib_goals(NewGoals, [], NewTail),
    NewStructure =.. [if, NewHead, NewTail],
    leave_trail(insert_before,
        [NewHead | NewGoals], ClauseName).

```

APPENDIX D

```

insert_before([NewHead | NewGoals], ClauseName,
  [], [NewStructure],
  LastUndo, []) :-
  last_traversal(next_clause, ClauseName),
  ib_goals(NewGoals, [], NewTail),
  NewStructure =.. [if, NewHead, NewTail],
  leave_trail(insert_before,
    [NewHead | NewGoals], ClauseName).
insert_before([NewHead | NewGoals], ClauseName,
  [], [NewStructure],
  LastUndo, []) :-
  no_last_traversal,
  ib_goals(NewGoals, [], NewTail),
  NewStructure =.. [if, NewHead, NewTail],
  leave_trail(insert_before,
    [NewHead | NewGoals], ClauseName).
insert_before(NewGoals, ClauseName,
  [Head | T], [NewHead | T],
  LastUndo, [Head | T]) :-
  Head =.. [Functor | _],
  comma_or_semicolon(Functor),
  ib_goals(NewGoals, Head, NewHead),
  leave_trail(insert_before, [g | NewGoals], ClauseName).
insert_before(NewGoals, ClauseName,
  [Head | T], [NewHead | T],
  LastUndo, [Head | T]) :-
  last_traversal(next_goal, ClauseName),
  ib_goals(NewGoals, Head, NewHead),
  leave_trail(insert_before, [g | NewGoals], ClauseName).
insert_before(NewGoals, ClauseName,
  [Head | T], [NewHead | T],
  LastUndo, [Head | T]) :-
  last_traversal(body, ClauseName),
  ib_goals(NewGoals, Head, NewHead),
  leave_trail(insert_before, [g | NewGoals], ClauseName).
insert_before(NewItems, ClauseName, [Head | Tail], [Head | Tail],
  LastUndo, LastUndo) :-
  Head =.. [if | _],
  pedir_error(ib, Structure).
insert_before(NewItems, ClauseName, [Head | Tail], [Head | Tail],
  LastUndo, LastUndo) :-
  Head =.. [Functor | _],
  comma_or_semicolon(Functor),
  pedir_error(ib, Structure).
insert_before(NewItems, ClauseName, [HofList | RestofList], NewList,
  LastUndo, [HofList | RestofList]) :-
  appl(NewItems, [HofList | RestofList], NewList),
  leave_trail(insert_before, NewItems, ClauseName).
insert_before(NewItems, ClauseName, Structure, Structure,
  LastUndo, LastUndo) :-
  pedir_error(ib, Structure).

```


APPENDIX D

```

ib_goals([], Goals, Goals).
ib_goals([Goal | [Functor | []]], [], Goal) :-
    comma_or_semicolon(Functor).
ib_goals([Goal | [], [], Goal]).
ib_goals([Goal | [Functor | []]], OldGoals, NewGoals) :-
    comma_or_semicolon(Functor),
    NewGoals =.. [Functor, Goal, OldGoals].
ib_goals([Goal1 | [Functor | RestGoals]], OldGoals, NewGoals) :-
    comma_or_semicolon(Functor),
    ib_goals(RestGoals, OldGoals, TempGoals),
    NewGoals =.. [Functor, Goal1, TempGoals].

/***** mv *****/
move_item([From, To], ClauseName,
    [Head | Tail], [NewHead | Tail],
    LastUndo, [Head | Tail]) :-
    From < To, Head =.. [Functor | Rest],
    comma_or_semicolon(Functor),
    move_goal_back(From, To, 1, [Head], NewHead, []),
    leave_trail(move_item, [From, To], ClauseName).
move_item([From, To], ClauseName,
    [Head | Tail], [NewHead | Tail],
    LastUndo, [Head | Tail]) :-
    From > To, Head =.. [Functor | Rest],
    comma_or_semicolon(Functor),
    move_goal_forward(From, To, 1,
        [Head], NewHead, Article),
    leave_trail(move_item, [From, To], ClauseName).
move_item([From, To], ClauseName, Head, NewHead,
    LastUndo, Head) :-
    From < To, move_back(From, To, 1, Head, NewHead, []),
    leave_trail(move_item, [From, To], ClauseName).
move_item([From, To], ClauseName, Head, NewHead,
    LastUndo, Head) :-
    From > To,
    move_forward(From, To, 1, Head, NewHead, Article),
    leave_trail(move_item, [From, To], ClauseName).
move_item([From, To], ClauseName, Structure, Structure,
    LastUndo, LastUndo) :-
    pedtit_error(mv, Structure).

move_goal_back(From, To, To,
    [Head], NewHead, [NewFunctor, Item]) :-
    Head =.. [Functor, Goal, RestGoals],
    comma_or_semicolon(Functor),
    NewRest =.. [NewFunctor, Item, RestGoals],
    NewHead =.. [Functor, Goal, NewRest].
move_goal_back(From, To, To,
    [Goal], NewGoals, [NewFunctor, Item]) :-
    NewGoals =.. [NewFunctor, Goal, Item].

```


APPENDIX D

```

move_goal_back(From, To, From,
  [Head], NewHead, Article) :-
  Head =.. [Functor, Goal, RestGoals],
  Current is From + 1,
  move_goal_back(From, To, Current,
    [RestGoals], NewHead, [Functor, Goal]).

move_goal_back(From, To, Current,
  [Head], NewHead, Article) :-
  Head =.. [Functor, Goal, RestGoals],
  NewCurrent is Current + 1,
  move_goal_back(From, To, NewCurrent,
    [RestGoals], NewRest, Article),
  NewHead =.. [Functor, Goal, NewRest].

move_goal_forward(From, To, From,
  [Head], RestGoals, [Functor, Goal]) :-
  Head =.. [Functor, Goal, RestGoals].
move_goal_forward(From, To, From,
  [Goal], [], [Functor, Goal]) :-
  goal_functor(Functor).
move_goal_forward(From, 0, 1,
  [Head], NewHead, Article) :-
  Head =.. [Functor, Goal, RestGoals],
  comma_or_semicolon(Functor),
  move_goal_forward(From, 0, 2,
    [RestGoals], NewRest, [NewFunctor, Item]),
  make_new_goals(Functor, Goal, NewRest, NewGoals),
  NewHead =.. [NewFunctor, Item, NewGoals].
move_goal_forward(From, 0, 1,
  [Goal], NewGoals, [NewFunctor, Item]) :-
  NewGoals =.. [NewFunctor, Goal, Item].
move_goal_forward(From, To, To,
  [Head], NewHead, Article) :-
  Head =.. [Functor, Goal, RestGoals],
  comma_or_semicolon(Functor),
  NewCurrent is To + 1,
  move_goal_forward(From, To, NewCurrent,
    [RestGoals], NewRest, [NewFunctor, Item]),
  make_new_goals(NewFunctor, Item, NewRest, NewGoals),
  NewHead =.. [Functor, Goal, NewGoals].
move_goal_forward(From, To, To,
  [Goal], NewGoals, [NewFunctor, Item]) :-
  NewGoals =.. [NewFunctor, Goal, Item].
move_goal_forward(From, To, Current,
  [Head], NewHead, Article) :-
  Head =.. [Functor, Goal, RestGoals],
  NewCurrent is Current + 1,
  move_goal_forward(From, To, NewCurrent,
    [RestGoals], NewRest, Article),
  make_new_goals(Functor, Goal, NewRest, NewHead).

```

APPENDIX D

```

make_new_goals(Functor, Goal, [], Goal).
make_new_goals(Functor, Goal, Rest, NewGoals) :-
    NewGoals =.. [Functor, Goal, Rest].

move_back(From, To, To,
    [Head | Tail], ReturningStructure, Article) :-
    app1([Head], Article, NewHead),
    app1(NewHead, Tail, ReturningStructure).
move_back(From, To, From,
    [Head | Tail], NewTail, Article) :-
    NewFrom is From + 1,
    move_back(From, To, NewFrom, Tail, NewTail, [Head]).
move_back(From, To, Current,
    [Head | Tail], [Head | NewTail], Article) :-
    NewCurrent is Current + 1,
    move_back(From, To, NewCurrent,
        Tail, NewTail, Article).

move_forward(From, To, From,
    [Head | Tail], Tail, [Head]).
move_forward(From, 0, 1,
    [Head | Tail], ReturningStructure, Article) :-
    move_forward(From, 0, 2, Tail, NewTail, Article),
    app1(Article, [Head], NewHead),
    app1(NewHead, NewTail, ReturningStructure).
move_forward(From, To, To,
    [Head | Tail], ReturningStructure, Article) :-
    NewCurrent is To + 1,
    move_forward(From, To, NewCurrent,
        Tail, NewTail, Article),
    app1([Head], Article, NewHead),
    app1(NewHead, NewTail, ReturningStructure).
move_forward(From, To, Current,
    [Head | Tail], [Head | NewTail], Article) :-
    NewCurrent is Current + 1,
    move_forward(From, To, NewCurrent,
        Tail, NewTail, Article).

/***** cp *****/
copy_item([From, To], ClauseName,
    [Head | Tail], [NewHead | Tail],
    LastUndo, [Head | Tail]) :-
    From < To, Head =.. [Functor | Rest],
    comma_or_semicolon(Functor),
    copy_goal_back(From, To, 1, [Head], NewHead, []),
    leave_trail(move_item, [From, To], ClauseName).
copy_item([From, To], ClauseName,
    [Head | Tail], [NewHead | Tail],
    LastUndo, [Head | Tail]) :-
    From > To, Head =.. [Functor | Rest],
    comma_or_semicolon(Functor),

```

APPENDIX D

```

copy_goal_forward(From, To, 1,
    [Head], NewHead, Article),
leave_trail(move_item, [From, To], ClauseName).

copy_item([From, To], ClauseName, Head, NewHead,
    LastUndo, Head) :-
    From < To, copy_back(From, To, 1, Head, NewHead, []),
    leave_trail(copy_item, [From, To], ClauseName).
copy_item([From, To], ClauseName, Head, NewHead,
    LastUndo, Head) :-
    From > To,
    copy_forward(From, To, 1, Head, NewHead, Article),
    leave_trail(copy_item, [From, To], ClauseName).
copy_item([From, To], ClauseName, Structure, Structure,
    LastUndo, LastUndo) :-
    peddit_error(cp, Structure).

copy_goal_back(From, To, To,
    [Head], NewHead, [NewFunctor, Item]) :-
    Head =.. [Functor, Goal, RestGoals],
    comma_or_semicolon(Functor),
    NewRest =.. [NewFunctor, Item, RestGoals],
    NewHead =.. [Functor, Goal, NewRest].
copy_goal_back(From, To, To,
    [Goal], NewGoals, [NewFunctor, Item]) :-
    NewGoals =.. [NewFunctor, Goal, Item].
copy_goal_back(From, To, From,
    [Head], NewHead, Article) :-
    Head =.. [Functor, Goal, RestGoals],
    Current is From + 1,
    copy_goal_back(From, To, Current,
        [RestGoals], NewRest, [Functor, Goal]),
    NewHead =.. [Functor, Goal, NewRest].
copy_goal_back(From, To, Current,
    [Head], NewHead, Article) :-
    Head =.. [Functor, Goal, RestGoals],
    NewCurrent is Current + 1,
    copy_goal_back(From, To, NewCurrent,
        [RestGoals], NewRest, Article),
    NewHead =.. [Functor, Goal, NewRest].

copy_goal_forward(From, To, From,
    [Head], Head, [Functor, Goal]) :-
    Head =.. [Functor, Goal, RestGoals].
copy_goal_forward(From, To, From,
    [Goal], Goal, [Functor, Goal]) :-
    goal_functor(Functor).
copy_goal_forward(From, 0, 1,
    [Head], NewHead, Article) :-
    Head =.. [Functor, Goal, RestGoals],
    comma_or_semicolon(Functor),
    copy_goal_forward(From, 0, 2,

```

APPENDIX D

```

        [RestGoals], NewRest, [NewFunctor, Item]),
    make_new_goals(Functor, Goal, NewRest, NewGoals),
    NewHead =.. [NewFunctor, Item, NewGoals].

copy_goal_forward(From, 0, 1,
    [Goal], NewGoals, [NewFunctor, Item]) :-
    NewGoals =.. [NewFunctor, Goal, Item].
copy_goal_forward(From, To, To,
    [Head], NewHead, Article) :-
    Head =.. [Functor, Goal, RestGoals],
    comma_or_semicolon(Functor),
    NewCurrent is To + 1,
    copy_goal_forward(From, To, NewCurrent,
        [RestGoals], NewRest, [NewFunctor, Item]),
    make_new_goals(NewFunctor, Item, NewRest, NewGoals),
    NewHead =.. [Functor, Goal, NewGoals].
copy_goal_forward(From, To, To,
    [Goal], NewGoals, [NewFunctor, Item]) :-
    NewGoals =.. [NewFunctor, Goal, Item].
copy_goal_forward(From, To, Current,
    [Head], NewHead, Article) :-
    Head =.. [Functor, Goal, RestGoals],
    NewCurrent is Current + 1,
    copy_goal_forward(From, To, NewCurrent,
        [RestGoals], NewRest, Article),
    make_new_goals(Functor, Goal, NewRest, NewHead).

copy_back(From, To, To,
    [Head | Tail], ReturningStructure, Article) :-
    app1([Head], Article, NewHead),
    app1(NewHead, Tail, ReturningStructure).
copy_back(From, To, From,
    [H | Tail], [H | NewTail], Article) :-
    NewFrom is From + 1,
    copy_back(From, To, NewFrom, Tail, NewTail, [H]).
copy_back(From, To, Current,
    [H | Tail], [H | NewTail], Article) :-
    NewCurrent is Current + 1,
    copy_back(From, To, NewCurrent,
        Tail, NewTail, Article).

copy_forward(From, To, From,
    [Head | Tail], Tail, [Head]).
copy_forward(From, 0, 1,
    [Head | Tail], ReturningStructure, Article) :-
    copy_forward(From, 0, 2, Tail, NewTail, Article),
    app1(Article, [Head], NewHead),
    app1(NewHead, Tail, ReturningStructure).
copy_forward(From, To, To,
    [Head | Tail], ReturningStructure, Article) :-
    NewCurrent is To + 1,
    copy_forward(From, To, NewCurrent,

```

APPENDIX D

```

        Tail, NewTail, Article),
    app1([Head], Article, NewHead),
    app1(NewHead, Tail, ReturningStructure).

copy_forward(From, To, Current,
    [Head | Tail], [Head | NewTail], Article) :-
    NewCurrent is Current + 1,
    copy_forward(From, To, NewCurrent,
        Tail, NewTail, Article).

/***** ch *****/
change_structure(NewStructure, ClauseName,
    Structure, NewStructure,
    LastUndo, Structure) :-
    leave_trail(change_structure,
        NewStructure, ClauseName).
change_structure(NewStructure, ClauseName,
    Structure, Structure,
    LastUndo, LastUndo) :-
    pedir_error(ch, Structure).

/***** cha *****/
change_argument([Num | [NewVal | []]], ClauseName,
    [Head | T], [Head | T],
    LastUndo, LastUndo) :-
    atomic(Head),
    write('No arguments present'), nl,
    pedir_error(cha, [Head | T]).
change_argument([Num | [NewVal | []]], ClauseName,
    [Head | T], [Head | T],
    LastUndo, LastUndo) :-
    functor(Head, Fn, N), Num > N,
    write('Argument number too high'), nl,
    pedir_error(cha, [Head | T]).
change_argument([Num | [NewVal | []]], ClauseName,
    [Head | T], [NewHead | T],
    LastUndo, [Head | T]) :-
    functor(Head, Fn, N),
    functor(NewHead, Fn, N),
    sub_arg(N, Num, NewVal, Head, NewHead),
    leave_trail(change_argument,
        [Num | NewVal], ClauseName).
change_argument(Args, ClauseName, Structure, Structure,
    LastUndo, LastUndo) :-
    pedir_error(cha, Structure).

sub_arg(0, _, _, _, _) :- !.
sub_arg(N, Num, New, Val, NewVal) :-
    arg(N, Val, OldArg),
    arg(N, NewVal, NewArg),

```

APPENDIX D

```

    subs(N, Num, New, OldArg, NewArg),
    N1 is N - 1,
    sub_arg(N1, Num, New, Val, NewVal).
subs(Num, Num, New, Old, New) :- !.
subs(_, _, _, Val, Val) :- !.

```

```

/***** chf *****/
change_this_functor([New], ClauseName,
    [Structure | T], [Structure | T],
    LastUndo, LastUndo) :-
    atomic(Val),
    write('No functor is present'), nl,
    pedit_error(chf, [Structure | T]).
change_this_functor([New], ClauseName,
    [Structure | T], [NewStructure | T],
    LastUndo, [Structure | T]) :-
    Structure =.. [Fuctor | Args],
    NewStructure =.. [New | Args],
    leave_trail(change_this_functor, [New], ClauseName).
change_this_functor([New], ClauseName, Structure, Structure,
    LastUndo, LastUndo) :-
    pedit_error(chf, Structure).

```

```

/***** chg *****/
change_goal([Num | [NewVal | [] ]], ClauseName,
    [Head | T], [Head | T],
    LastUndo, LastUndo) :-
    atomic(Head),
    write('No arguments present'), nl,
    pedit_error(chg, [Head | T]).
change_goal([Num | [NewVal | [] ]], ClauseName,
    [Head | T], [NewHead | T],
    LastUndo, [Head | T]) :-
    Head =.. [Fn | _],
    comma_or_semicolon(Fn),
    ch_goal(Num, 1, NewVal, Head, NewHead),
    leave_trail(change_goal, [Num | NewVal], ClauseName).
change_goal(Args, ClauseName, Structure, Structure,
    LastUndo, LastUndo) :-
    pedit_error(chg, Structure).

```

```

ch_goal(Num, Num, NewVal, Head, NewHead) :-
    Head =.. [Fn, Goal, RestGoals],
    comma_or_semicolon(Fn),
    NewHead =.. [Fn, NewVal, RestGoals].
ch_goal(Num, Current, NewVal, Head, NewHead) :-
    Head =.. [Fn, Goal, RestGoals],
    comma_or_semicolon(Fn),
    NewCurrent is Current + 1,
    ch_goal(Num, NewCurrent, NewVal, RestGoals, NewGoals),

```

APPENDIX D

NewHead =.. [Fn, Goal, NewGoals].

```

/***** rem *****/
remove([Name | []], ClauseName, Structure, [],
      LastUndo, Structure) :-
    asserta(buffer(Name, Structure)),
    leave_trail(remove, [Name], ClauseName).
remove(Args, ClauseName, Structure, Structure,
      LastUndo, LastUndo) :-
    pedir_error(rem, Structure).

/***** repl *****/
replace([Name | []], ClauseName, Structure, NewStructure,
      LastUndo, Structure) :-
    buffer(Name, NewStructure),
    same_format(Structure, NewStructure),
    leave_trail(replace, [Name], ClauseName).
replace([Name], ClauseName, Structure, Structure,
      LastUndo, LastUndo) :-
    pedir_error(repl, Structure),
    buffer(Name, NewStructure),
    write('Buffer: '),
    write(Name), nl,
    write('Structure: '),
    pp(NewStructure), nl.
same_format(S,N).

/***** un *****/
undo([], ClauseName, Structure, Structure,
      LastUndo, LastUndo) :-
    undocommand(CurrentCommand),
    invalid_undo(CurrentCommand),
    pedir_error(un, Structure).
undo([], ClauseName, Structure, Structure,
      LastUndo, LastUndo) :-
    undocommand(CurrentCommand),
    traversal_command(CurrentCommand, _, NewCommand),
    remove_old_undo,
    asserta(cmd_list([NewCommand])).
undo([], ClauseName, Structure, LastUndo,
      LastUndo, Structure) :-
    remove_old_undo.
undo([], ClauseName, Structure, Structure,
      LastUndo, LastUndo) :-
    pedir_error(un, Structure).

remove_old_undo :-
    retract(undocommand(X)).
remove_old_undo.

```

APPENDIX D

```

/***** buffer *****/
create_buffer([Name | Contents], ClauseName,
    Structure, Structure,
    LastUndo, LastUndo) :-
    asserta(buffer(Name, Contents)).
create_buffer(Args, ClauseName, Structure, Structure,
    LastUndo, LastUndo) :-
    pedir_error(buffer, Structure).

/***** display *****/
display_buffers([], ClauseName, Structure, Structure,
    LastUndo, LastUndo) :-
    write('List of buffer names: '), nl,
    write_buffer_names.
write_buffer_names :-
    buffer(Name, X),
    write(' '),
    write(Name), nl,
    fail.
write_buffer_names.

/***** contents *****/
buffer_contents([Name], ClauseName, Structure, Structure,
    LastUndo, LastUndo) :-
    buffer(Name, Contents),
    write('Contents of buffer '),
    write(Name),
    write(' are: '), nl,
    pp(Contents).
buffer_contents([Name | []], ClauseName,
    Structure, Structure,
    LastUndo, LastUndo) :-
    write('Buffer '),
    write(Name),
    write(' does not exist. '), nl,
    pedir_error(contents, Structure).

/***** chdelim *****/
change_delimiter([New | []], ClauseName,
    Structure, Structure,
    LastUndo, LastUndo) :-
    retract(delim(X)),
    asserta(delim(New)).
change_delimiter(Args, ClauseName, Structure, Structure,
    LastUndo, LastUndo) :-
    pedir_error(chdelim, Structure).

```


APPENDIX D

```

/***** rpt *****/
repeat([], ClauseName, Structure, Structure,
      LastUndo, LastUndo) :-
    last_command(Command, Arglist),
    asserta(cmd_list([Command | Arglist])).
repeat([], ClauseName, Structure, Structure,
      LastUndo, LastUndo) :-
    write('No command to repeat'), nl,
    pedit_error(rpt, Structure).

/***** defcmd *****/
define_command([Name | Args], ClauseName,
              Structure, Structure,
              LastUndo, LastUndo) :-
    asserta(new_command(Name, Args)).
define_command(Args, ClauseName, Structure, Structure,
              LastUndo, LastUndo) :-
    pedit_error(defcmd, Structure).

/***** displcmd *****/
display_commands([], ClauseName, Structure, Structure,
                LastUndo, LastUndo) :-
    write('List of command names: '), nl,
    write_command_names.
write_command_names :-
    command(Name, Args),
    write(' '),
    write(Name), nl,
    fail.
write_command_names.

/***** command *****/
command_contents([Name], ClauseName,
                Structure, Structure,
                LastUndo, LastUndo) :-
    command(Name, Contents),
    write('Contents of command '),
    write(Name),
    write(' are: '), nl,
    pp(Contents).
command_contents([Name | []], ClauseName,
                Structure, Structure,
                LastUndo, LastUndo) :-
    write('Command '),
    write(Name),
    write(' does not exist. '), nl,
    pedit_error(command, Structure).

/* miscellaneous routines and rules */

```

APPENDIX D

```

/* traversal_command -                               */
/*   Parameters:                                     */
/*     Command Name                                   */
/*     Direction of Command                           */
/*     Opposite Traversal                             */
traversal_command(top, up, next_clause).
traversal_command(next_clause, down, previous_clause).
traversal_command(previous_clause, up, next_clause).
traversal_command(head, down, start).
traversal_command(body, down, start).
traversal_command(start, up, head).
traversal_command(next_goal, down, previous_goal).
traversal_command(previous_goal, up, next_goal).
traversal_command(down, down, up).
traversal_command(up, up, down).
traversal_command(right, down, left).
traversal_command(left, up, right).

modification_command(delete_structures).
modification_command(insert_after).
modification_command(insert_before).
modification_command(move_item).
modification_command(copy_item).
modification_command(change_structure).
modification_command(change_argument).
modification_command(change_this_functor).
modification_command(change_goal).
modification_command(remove).
modification_command(replace).

invalid_undo(file).
invalid_undo(find_argument).

```

APPENDIX D

```

leave_trail(Command, Arglist, ClauseName) :-
    traversal_command(Command, down, _),
    add_traversal(Command, ClauseName),
    new_undo(Command, Arglist),
    add_command(Command, Arglist).
leave_trail(Command, Arglist, ClauseName) :-
    traversal_command(Command, up, _),
    new_undo(Command, Arglist),
    add_command(Command, Arglist).
leave_trail(Command, Arglist, ClauseName) :-
    modification_command(Command),
    add_changed(ClauseName),
    new_undo(Command, Arglist).
leave_trail(Command, Arglist, ClauseName) :-
    new_undo(Command, Arglist).

new_undo(Command, Arglist) :-
    invalid_undo(Command),
    asserta(last_command(Command, Arglist)).
new_undo(Command, Arglist) :-
    remove_old_undo,
    asserta(undo_command(Command)),
    asserta(last_command(Command, Arglist)).

add_changed(Name) :- changed(Name).
add_changed(Name) :- asserta(changed(Name)).

pedit_error(Command, Structure) :-
    write('Can not perform '),
    write(Command),
    nl,
    pp(Structure),
    nl, nl,
    retract_all(cmd_list(_)).

count(Command, Structure) :-
    count_them(0, Command, Total),
    write_them(Command, Total, Structure).
count_them(Num, Command, Total) :-
    retract(counter(Command)),
    NewNum is Num + 1,
    count_them(NewNum, Command, Total).
count_them(Total, _, Total).

write_them(Command, 0, Structure) :-
    pedit_error(Command, Structure).
write_them(Command, Total, _) :-
    write(' '),
    write(Total),
    write(' repetition(s) '),
    nl.

```

APPENDIX D

```

retract_all(X) :- retract(X), fail.
retract_all(X) :- retract((X:-Y)), fail.
retract_all(_).

add_traversal(Name, ClauseName) :-
    asserta(traversal(ClauseName, Name)).

remove_traversal(Name, ClauseName) :-
    retract(traversal(ClauseName, Name)).
remove_traversal(_, _).

last_traversal(Name1, ClauseName) :-
    retract(traversal(ClauseName, Name2)),
    asserta(traversal(ClauseName, Name2)),
    !,
    Name1 = Name2.

no_last_traversal :-
    retract(traversal(CN, N)),
    !, fail.
no_last_traversal.

add_command(Name, []).
add_command(Name, [*]) :-
    asserta(cmd_list([Name, *])).
add_command(Name, [Num]) :-
    integer(Num),
    NewNum is Num - 1,
    asserta(cmd_list([Name, NewNum])).
add_command(_, _).

app1([], X, X).
app1([A|B], C, [A|D]) :- app1(B, C, D).

empty([]).
not_null([]) :- !, fail.
not_null(_).
comma_or_semicolon(',', ' ').
comma_or_semicolon(';', ' ').
goal_functor(' ', ' ').
not_if(if) :- !, fail.
not_if(_).
eofline(EofLine) :- name(EofLine, [10]).

```

PEDIT -
A Resident Structure Editor
for PROLOG

by

SANDRA LEE DUFFY

B.S. University of Illinois, 1977

AN ABSTRACT OF A MASTER'S REPORT

submitted in partial fulfillment of the

requirements for the degree

MASTER OF SCIENCE

Department of Computer Science

KANSAS STATE UNIVERSITY
Manhattan, Kansas

1985

PEDIT -
A Resident Structure Editor
for PROLOG

This paper presents PEDIT - a residential structure editor implemented in PROLOG. PEDIT allows the user to manipulate the PROLOG data base without exiting the PROLOG interpreter. This facility will greatly facilitate the interactive testing process of PROLOG programs. Savings will be incurred by using PEDIT, especially when dealing with a large PROLOG program. The costs of repeatedly exiting and entering the interpreter plus the cost of reloading the PROLOG file(s) are saved.

Chapter One of this paper details the motivation for developing PEDIT. A comparison of editor types and existing editors is also included. Chapters Two, Three and Four describe the requirements, design and implementation, respectively, of PEDIT. Chapter Five contains conclusions and suggests extensions to PEDIT. The Appendices provide a glossary of PROLOG terminology, a brief summary of the available commands, the User's Manual and the source code.