EPILOG: The Computational System for Episodic Logic

USER'S GUIDE

Stephanie Schaeffer

Chung Hee Hwang

John de Haan

Lenhart K. Schubert

August 1993

Revised September 2000

Prepared for Boeing Computer Services, Seattle, Washington

Under Purchase Contract W-278258

Contents

Chapter 1

Use of EPILOG

1.1 Introduction

EPILOG is an inference engine designed to handle the representation and low level reasoning necessary for natural language understanding. Although it was designed with this purpose in mind, the representation and inference it does can be used for other domains with similar requirements. It uses an extended first order logic called **episodic logic** (**EL**) (Hwang and Schubert) which allows propositional attitudes, unreliable generalizations, and other non-standard constructs, including ones involving events, actions, facts, kinds and donkey sentences. Episodic sentences are represented using episodic variables, which can be used to capture temporal and causal relationships. Axiom schemas help to control the number of rules required, as well as represent narrative and meaning postulate information. The rules of inference include probabilistic versions of deduction rules resembling the forward and backward chaining rules in expert systems. Special inference methods (specialists) for certain specific domains assist the general inference mechanism to increase overall efficiency. [1]

EPILOG maintains and answers questions about story knowledge given to it in episodic logic. Input-driven inference is done on input of story information (i.e. it figures out the "consequences" of the input story facts), and yes/no questions and some wh-questions can be answered. The system can also generate English responses to questions, and repeats input facts and rules and inferences in English as well. Note that, however, the current version of **EPILOG** has limited inferencing capability – it does not perform some narrative or simulative inferences yet, and some other processes are not yet fully handled.

The system is currently running in Allegro Common Lisp (version 5.0). Some knowledge of Lisp will be helpful in using the system, but is not essential.

If you are a new user, read the "New User's Tutorial" first for a painless introduction to the system. In this manual (the User's Guide), Chapter 1 contains instructions on starting the system up, and general instructions such as how to trace actions, get help, or tweak parameters. Chapter 2 contains details on the logical syntax accepted by the system. The rest of the chapters deal separatly with the main system operations. Chapter 3 shows how to get input (hierarchy and formulas) into the system, and describes the assertion process. Chapter 4 shows how to query the system for information, and how to ask questions, and the question answering process. Chapter 5 describes the specialists which assist the system, how to invoke them, when and where they apply, and how to control them. Chapter 6 describes the response

[1] For a typical user, this manual contains enough information to use the system. The References section at the end of this manual contains a list of reports which describe episodic logic in detail, including the theoretical underpinnings.

generator, including how to put translation and lexical information into the system, and how the response generator works. Chapter 7 contains topics for use by experienced users. Appendix 1 contains a small glossary of some of the terms used in the manual;

EPILOG users may have varying backgrounds and may not necessarily be familiar with all of the terms used. Appendix 2 contains a small reference list of papers which may be helpful in understanding EPILOG.

There are several places in the manual where **EPILOG** is compared to a system called **ECoNet** . **ECoNet** (Miller & de Haan, 1987 & 1988) was the predecessor of **EPILOG** , and many of the techniques used there have been incorporated into **EPILOG** . **EPILOG** is different from **ECoNet** in that it supports an extension of predicate logic and input driven inferences and that it is natural deduction based rather than resolution based.

1.2 Getting Started

The root directory of the **EPILOG** distribution is called `EPILOG`. To use the system, start lisp, and then load the file `EPILOG/epi/epi`. (Before using it for the first time, compile the system by running lisp and loading `EPILOG/epi/compile-all.lisp`.)

If the directory where lisp was started contains a file called *epi-init.lisp* , that file is automatically loaded whenever **EPILOG** is started up (this can be avoided by setting the variable `*load-user-epi-init*` to `nil`). This file should contain loads of hierarchies you commonly use, tweak parameters to the values you like to use, and set up tracing for the items you usually use. For example, the contents could be:

```
(load "eg.hier")
(load "eg.indicate")
(load "eg.parts")
(load "eg.lex")
(load "eg.trans")
(tweak '*qa-iterations* 20)
(trace-item 'qa 'entry)
```

EPILOG may be used directly from the **epilog** package: evaluate `(in-package epilog)` after loading the system. A more powerful interface which also allows escapes to the unix shell is available as well - this is the **epishell** . If you are using **EPILOG** from another system which might have name conflicts with it, you can use the user interface **epiuser** instead. A **(use-package 'epiuser)** will import only the main **EPILOG** routines described in this manual. When using **epiuser** commands, all input will be interned into the **epilog** package automatically.

Once you have **EPILOG** started up, you will want to assert information (with the **add -hier** , **story** and **kn** functions), and ask questions (with the **q** function). These operations are discussed in detail in Chapters 3 and 4. The syntax for the formulas involved in described in Chapter 2. The rest of this chapter describes some of the general **EPILOG** commands needed to get help, print information, trace system actions, etc.

1.3 Help

The help facility enables you to get on-line help for specific commands and some topics. This help generally contains exactly what is in the user manual - the purpose of the command, its arguments and

what they mean, and any specific comments about it.

(help *&optional topic*) [*function*]

Purpose: Provides help about the usage of the system.

Syntax: *topic* may be either a command name, or another topic for which help exists. To find out these topics, do **(help 'topics)** , or just **(help)** . If no topic is specified, a summary of the available commands is printed.

Examples:

(help)

(help 'knowledge)

1.4 The EpiShell

The EpiShell is an optional command line interpreter that can be used within the EPILOG environment, as an alternative to the usual Lisp top-level read-eval loop. The main purpose of the shell is to make it simple to use EPILOG as an experimental tool, by creating an interactive environment in which Lisp, EPILOG the file system, and operating system tools are all readily available. Figure 1.1 describes the shell's top-level read-eval loop.

```
loop
    read the next command

    perform standard input/output file redirection as indicated.

    if the command is a list, then
       evaluate as a Lisp form

    else if the first symbol on the line matches a command or
             synonym known to the shell, or is an abbrevation
             of any command or synonym, then

       evaluate the Lisp function associated with the command
       passing it the rest of the command line as implicitly
       quoted arguments

    else if there is only one symbol on the line then
       attempt to display its value

    else
       assume the first symbol exactly matches the name of a
       Lisp function, and evaluate that function, implicitly
       quoting the arguments on the rest of the line

    endif

until end of file
```

Figure 1.1. Top level read-eval loop of the EpiShell

1.4.1 Starting up an EpiShell

From Lisp, the way to start up a shell is to invoke the function **esh** .

(esh *&optional input-spec profile-filename*) [*function*]

Purpose: Starts a new Ep.SHell.

Syntax: If *input-spec* is a stream, the shell will use the stream as its standard input, exiting upon end-of-file on the stream. Otherwise, if *input-spec* is a symbol, it's assumed to be the name of a file containing shell commands, or, if *input-spec* is a string, the shell will execute the single command specified by the string.

The profile file is a handy way to make the shell look the way you want it to. It's simply a file of shell commands that gets executed before control is passed to the terminal. For example, a profile file might consist of a single line that sets the prompt to your favorite string:

set prompt "==> "

By default, **esh** assumes *profile-filename* is **epishell.pro** . If the file does not exist, the option is ignored.

Examples:

(esh)

(esh "! ls")

(esh 'in-file)

1.4.2 The EpiShell environment

While you are in the shell, the top-level read-eval loop will continually prompt you to enter either a command or a Lisp expression. Commands are entered one to a line: the shell uses the newline character (i.e., the **Return** key) to detect the end of a command. However, it ignores newlines embedded inside lists, and thus, although commands in themselves cannot span multiple lines, command arguments that are lists can, as can stand-alone lists which are to be evaluated as Lisp expressions. The shell exits when it detects an end of file on its standard input. In the Sun implementation, end of file is signaled by typing **Control-D** (i.e., pressing the **Control** and the **D** keys at the same time).

1.4.2.1 File redirection

EpiShell commands can obtain additional input from the **standard input** and direct output to the **standard output** . By default, the standard input and output are the same as for the shell (which is usually the terminal), but they can be re-directed to named files by using the special symbols < , > , and >> . For example,

display >time.save time-info -full t1

puts the output from the **display** command in the file **time.save** . Similarly, < *filename* takes input from a file, and >> *filename* appends to an existing file (which it creates if it does not in fact exist). The shell displays symbol values and function results onto the standard output, so these too can be redirected to a file. In the following, the value of the symbol **a** is redirected into a file:

a >a.val

It is also possible to redirect the standard ouput into a pipe, as follows:

display hier | more

The implementation of pipe in the Ep.SHell is done using a temporary file. Specifically, given a command of the form *cmd [args]* | os-cmd , the shell collects all output from *cmd* into a temporary file, and then executes the operating system command **cat** *tmp-file* | *os-cmd* in an operating system shell (**/bin/csh**).

If < *filename* appears on a line by itself, then the shell temporarily redirects its own input to that file, using the contents of that file as a shell *script* .

Sometimes you might want to include the contents of a file somewhere in the middle of a list or command. The < mechanism cannot be used to this, but it can be done by using the syntax **@** *filename* , which includes the contents of that file exactly as if you had typed them in at that point. This mechanism can be embedded, so files which are included can themselves include other files. Line breaks in files included using **@** are interpreted as ordinary white-space, so this facility can also be used to include a command which spans more than one line. For example,

@long-cmd

would interpret all of the contents of the file **long-cmd** as a single command, even if the command spanned several lines in the file.

The last of the special characters that the shell recognizes is the **;** character, indicating the start of a comment that is assumed to consume the rest of the line. Comments can appear anywhere, including files which are read using the < or **@** mechanisms.

1.4.2.2 Using a default function

The shell's normal reaction to a list standing on its own is to evaluate it as a Lisp form. This behavior can be altered by using the command **set default-function** *function-name* , which causes the shell to evaluate **(** *function-name list* **)** whenever it encounters a stand-alone list. For example,

```
E> set default-function story
D> (LRRH girl)
                        ; equivalent to (story '(LRRH girl))
                        ;              or story (LRRH girl)
D> (W wolf)
                        ; equivalent to story (W wolf)
D> set default-function
E>
```

As the example shows, the shell uses a different prompt when in this mode. This facility can be turned off by invoking the **set default-function** command with no arguments.

1.4.3 EpiShell built-in commands

A shell command is invoked by simply typing its name in response to the shell prompt, followed by its arguments, and terminated by a newline. Abbreviations (of at least two characters) are recognized, so

if you're a slow typist you don't have to type the full name of a command. However, you should be aware that non-unique abbreviations are assigned on a first-come first-served basis. The command **show command** *abbrev* will tell you which command the abbreviation *abbrev* will invoke.

help *cmd-name*
This command gives on-line help for the command *cmd-name* , which can be any command, command synonym, or abbreviation recognized by the shell.

set *option*
Where *option* is one of:

command *cmd-name* {*synonym*}*
Adds a new command to the current shell. After setting a new command, typing *cmd-name* or any of the synonyms will invoke the Lisp function *cmd-name* . Any abbreviation of *cmd-name* or its synonyms that is not already an abbreviation of an existing command will also invoke the same function. (You can override an existing abbreviation by explicitly making it a synonym of the new command.)

If you're not sure what function a command or abbreviation is bound to, you can display the Lisp function that will be invoked by the command *cmd-str* by typing **show command** *cmd-str* . A complete list of all commands and their synonyms can be displayed by typing **show commands** . The **apropos** command can also be used to do a quick lookup of selected commands.

comment-char character
Changes the shell comment character to *character* . If no *character* is specified, the comment feature will be disabled.

default-function function-name
Defines the name of the function to invoke when the shell is given a list standing on its own. The list will be passed as the single argument to the function *function-name* . If no *function-name* is specified, this feature is disabled.

default-prompt *prompt-string*
Sets the prompt to use when a *default-function* (see above) has been set.

default-syntax *character*
Sets the syntax of a character back to the Lisp default. (see **set symbol-syntax**).

[no]echo
Instructs the shell to echo (or not) command files included using the < *filename* facility.

include-char { character}
Changes the shell include (read from a file) special character. If no *character* is specified, the include feature will be disabled.

prompt *prompt-string*
Changes the shell prompt to *prompt-string* .

pipe-char character
Changes the shell pipe (to an operating system command) special character. If no *character* is specified, the pipe feature will be disabled.

put-char { character}
Changes the shell put (to a file) special character. If no *character* is specified, the put feature will be disabled.

symbol-syntax *character*
Sets the syntax of *character* so that it will always be read as a single, stand-alone symbol. For example, by default the shell sets this syntax for the character **!** . The symbol **!** is also a synonym for the command **os-shell** , so it possible to type:

```
!ls *.lisp
```

and have the shell parse **!** as a command to invoke the operating system shell with the os command **ls** . Note that no space is needed after the **!** . To revert to the usual Lisp syntax for these this character (or any others that have been set using this command), use the command **set default-syntax** .

take-char { character}
Changes the shell take (from a file) special character. If no *character* is specified, the take feature will be disabled.

trace-output { filename}
Opens the file *filename* and sets the value of the global variable ***esh-trace-output*** to the corresponding file stream. If no filename is specified, trace output reverts to the standard output.

show *option*
Where *option* is **version** or anything that can be **set** .

apropos *search-string*
Gives a one-line description of all shell commands, synonyms, and document strings that contain *search-string* as a substring.

cat { filename}
Concatenates all the named files together by printing them in order onto the standard output. If no *filename* is specified, input is taken from the standard input instead. This rather innocuous-looking command can be used for several useful file operations. For example,

```
cat >newfile
```

creates the file **newfile** with the initial contents taken from the terminal (use **Control-D** to signal end of file). The command

```
cat newfile
```

can then be used to display the contents of the file, and the command

```
cat newfile >another-file
```

to copy the them to another file.

cd { directory } Changes the current file directory to *directory* . If *directory* contains the string "..", all of *directory* should be enclosed in double quotes, to avoid confusing the Lisp parser.

drib { filename}
If the *filename* argument is present, everything that appears on the terminal will also be dribbled to the file *filename* . **drib** with no arguments turns off this feature.

echo { argument } *
Echos *argument(s)* onto the standard output. Arguments are converted to lowercase, unless enclosed in double quotes. One possible use of this command (or its synonym, %) is to add comments to a file of EpiShell commands; these comments will then be displayed when the file is presented as input to the shell. For example, if the file **test.data** contains

```
% Setting up test checkpoint
checkpoint test
    .
    .
    .
```

then "Setting up test checkpoint" will appear when the shell is invoked (recusively) by giving the command **esh test.data** to the current shell.

esh { filename | *command* }
Starts up a sub-shell. In a sub-shell, the shell environment reverts back to the default—all of the **set** options (with the exception of **set command**) assume their usual start-up values. A profile file is not executed. By default, the sub-shell reads commands from the current standard input, but it can also get them from the file *filename* or from the single *command* argument (which must be enclosed in quotes, to distinguish it from a file name).

os-shell { command } { command-args } *
Start up an operating system shell. In the Sun implementation, this starts up **/bin/sh** , with standard input, output, and error output bound to the corresponding streams in the current shell. If *command* and *command-args* are present, they are converted to lowercase (unless they are enclosed double quotes), and then passed to **/bin/sh** using the **-c** option of that shell.

The synonym ' **!** ' can be also be used to invoke this command.

os-csh { command } { command-args } *
Start up the operating system **/bin/csh** shell.

shell commands
The following commands have also been set up as 'aliases':

Alias	Interpretation
emacs *args*	os-csh xemacs *args*
less *args*	os-shell less -dm *args*
lpq *args*	os-shell lpq *args*
ls *args*	os-shell ls *args*
paf *args*	os-shell paf -format P *args*
pwd *args*	os-shell pwd *args*
vi *args*	os-shell vi *args*

1.5 Display

In a large system with a number of parts, it is difficult for the user to keep track of all the routines he needs to use to print out various bits of information. To help in this, and to provide a uniform interface, the display command was set up. This command allows you to ask for specific information to be printed without being overly concerned with the parameter specification.

(display *&optional args*) [*function*]

Purpose: Prints information about a specific concept/topic from the topical classification table.

Syntax: Among the arguments may be a **-brief** (**-b**) or **-full** (**-f**) flag - this will be stripped off immediately. Of the remaining arguments, the first one should be an indicator of what is to be displayed (a topic), and the rest are arguments to that display function. These display functions are described throughout the manual, and include functions to print out knowledge about a particular individual or predicate, the current state of the question answering mechanism, the inference path used to answer a question or generate an input-driven inference, and specialist domain information. Calling **display** with no arguments will print the allowed "topics". If no topic is given, and the arguments are all tweakable parameters or trace values, their descriptions and values will be printed. Otherwise the first argument is considered to be a concept, and information about that concept is retrieved and displayed. You can also get the display command to look for desired options for you. If a **-key** (**-k**) flag is among the arguments, then the lisp function *apropos* is used on each argument, and the valid options which contain those arguments are displayed. If **tweak** is specified as well, the tweakable parameters which match the given arguments will be displayed. If **trace** is specified, the traceable values that match are displayed. If **display** is specified, the display options that match are displayed (the default). These may be used in any combination.

Examples:

(display 'lrrh '-full)

displays all information entered about **lrrh** , including belief subnets.

(display 'event-info '-b)

displays brief information on events in the time specialist.

(display 'time-info '-full 't1)

displays all information on the time point *t1* in the time specialist.

(display '-k 'time)

displays the valid display options that have *time* in their names.

(display '-k 'tweak 'trace 'qa '-f)

displays the valid tweakable parameters and trace values that have *qa* in their name. The -f flag also causes the current and default values of
the tweakable parameters to be printed.

Remarks: Note that the examples using **time-info** and **event-info** will only display something if the time specialist is active, and if some information has been passed to it.

1.6 Tracing and System Actions

These routines make it simpler to watch the system operate and to debug. Each operation of the system has a set of trace values associated with it (which are described in the section for the operation, as well as in the Quick Reference Guide). When tracing is turned on for a trace value, additional information is printed out during the operation it belongs to. Trace values are also associated with the specialists so that you can watch them as well. Tracing may be turned off or on for individual trace items or for all. Note that the Lisp functions **trace** and **untrace** have been advised so that when called with quoted values, they call **trace-item** or **untrace-item** respectively for the quoted values (unquoted items are

handled by Lisp).

The system starts out automatically tracing input driven inferences made (**forward**), and question answers (**qa**). These can be turned off, but it may be difficult to see what is happening then. If you want all printing to stop temporarily, you can tweak the ***print*** flag to nil, and then back to t when you want to start printing again. This will stop all **EPILOG** output.

(trace-item *items*) [*function*]

Purpose: To turn on tracing for selected items or for all.

Examples:

(trace-item 'forward 'rules)

Remarks: If no items are given, all traceable items will be traced.

(traceable *item description trace-values*) [*function*]

Purpose: To indicate that a given name is something which may be traced. This item may be a compound item of several other traceable items.

Syntax: *item* becomes a traceable value. *description* should be a string indicating what this trace value does. *trace-values* is a list of the traceable items that should have tracing turned on whenever this item is traced.

Examples:

(traceable 'qa "interesting stuff about question-answering" '(qa-time qa-test qa-eval qa-access qa-success unify))

Remarks: This can be useful if you have a set of trace items that you would like to be able to turn off and on easily. For example:

(traceable 'my-traces "stuff I'm interested in" '(forward-details entry-class))

(trace-all) [*function*]

Purpose: To turn on tracing for all traceable items.

Examples:

(trace-all)

(untrace-item *items*) [*function*]

Purpose: To turn off tracing for selected items or for all.

Examples:

(untrace-item 'rules 'classification)

Remarks: If no items are given, tracing will be stopped for all currently traced items.

(untrace-all) [*function*]

Purpose: To turn off tracing for all traced items.

Examples:

(untrace-all)

There is a display option **trace** which can be used to print out trace values and their descriptions:

(display 'trace *items* **)**

If no *items* are specified, all trace items will be displayed. Otherwise only the valid trace values in *items* will be printed. If the **-key** or **-k** flag is used, all valid trace values whose names contain one of *items* will be printed. The **-full** and **-brief** flags have no effect here.

1.7 Tweaking System Parameters

These routines make it easier to find and change system parameters. The parameters which may be changed are described in the sections about the operations they affect, and are summarized in the Quick Reference Guide. Parameters may be changed back to a preset default as well.

(tweak *item &optional newvalue*) [*function*]

Purpose: To change the value of a system parameter.

Syntax: Note that *item* should be quoted. If *newvalue* is not given, *item* is set back to its default.

Examples:

(tweak '*qa-iterations* 1)

Remarks: The tweakable parameters are described under the areas of the system they deal with in this manual. They may be displayed by calling **tweak** with no arguments. Limited checking is done on *newvalue* to ensure that it is a valid value for *item* .

There is a display option **tweak** which may be used to print out information on all or specific tweakable parameters:

(display 'tweak *items*)

If no *items* are specified, all are printed. If specific items are given, only they are printed. If **-full** or **-f** is specified, the current and default values of the parameter, as well as its description are printed. With **-brief** or **-b** , only the description is printed. If **-key** or **-k** is specified, any tweakable parameters which have any of *items* in their names are printed.

1.8 Checkpointing and Retraction

These routines allow you to set checkpoints during a session and to retract any formulas or changes to hierarchies entered after a checkpoint. This does not include changes to system parameters or the list of items being currently traced. Note that retraction always removes things in reverse, starting with the latest formula entered.

There are two main checkpointing/retraction methods. One is to use named checkpoints, and to retract to those named points. These may be nested, so that you can retract as much or as little as you

like. The other is to use numbers, and retract the last number of formulas entered. The two methods may be mixed so that you can retract a number of formulas while still keeping track of a named checkpoint.

(checkpoint *&optional item*) [*function*]

Purpose: To set up a named checkpoint, or to start a revolving checkpoint to enable retraction of formulas.

Syntax: If *item* is a number, a revolving checkpoint is set up which allows retraction of up to the last *item* formulas. If *item* is a string, or is absent, a checkpoint is set up which allows retraction of any formulas entered after the checkpoint, regardless of how many there are. The function returns a symbol which can later be given to the **retract** function. If *item* is a string, it is included in the symbol's name, to make examining the checkpoint stack more informative.

Examples:

(checkpoint 'start)

(checkpoint 5)

(retract *item*) [*function*]

Purpose: To retract all formulas entered since a given checkpoint, or to retract a certain number of formulas (in reverse order).

Syntax: If *item* is a number, the last *item* formulas are retracted, if possible. Forward inferences done when a formula is entered are also retracted. If *item* is a symbol previously returned by the **checkpoint** function, all formulas since that checkpoint was set up are retracted.

Examples:

(retract 'start)

(retract 2)

Remarks: You cannot retract more formulas than were entered since the **checkpoint** command. Also, once you have retracted the maximum number of formulas being keep in the revolving checkpoint, you cannot retract anymore, even if that many have been entered. For example, if a *(checkpoint 4)* command is issued, and some forumlas entered, a *(retract 3)* will work fine, but if another *(retract 3)* command is issued, only one more can be retracted. To display the checkpoint names already given, use the **display** command with topic *checkpoint* - i.e. **(display 'checkpoint)**
To permanently remove the named checkpoint after the retraction is done, use **(retract** *name* :remove t) . This stops checkpointing for that name.

1.9 Permanent Memory

Sometimes it is desirable to save the formulas and inferences made up to a given point, and reload them later, saving the trouble and time of re-doing the loading and inferences. One method of doing this is to use the Lisp *disksave* facility, but that requires a great deal of storage space. An alternative method is handled by EPILOG. All current formulas, and their properties, and constants and variables and their

properties may be save in a relatively small file, and reloaded quickly from that file. Currently only the information in the main system itself is saved - information in the specialists own storage media is not saved. What this means is that information which has not been sent to a specialist (like the time specialist) for storage can be quickly and easily saved and restored without changing the system's performance, although questions and inferences that rely on that specially stored information will not work.

The flag ***memory-load-specs*** can be used to indicate that the specialist information should be rebuilt when the permanent memory file is loaded in. If set to *t* (the default), each formula will be sent to any applicable specialists to enter into their domain. Although the newly built specialist domains will have a slightly different internal arrangement than the version the memory was stored from (because the formulas are entered in a different order), the results when question answering should be the same. This is a little slower than just loading the formulas themselves (with ***memory-load-specs*** *nil*), but still much faster than loading the entire test from scratch, including all input-driven inferencing.

This works especially well for a system with a large, fixed set of rules - these can be easily stored in a permanent memory file and loaded quickly. The storage mechanism does not store hierarchies or topic indicators - it is assumed that the identical hierarchies and topic indicators will be reloaded before the permanent memory file is called in. Any specialists which were active at the time of the storage are also activated. Parameter values are not saved, nor are trace values.

(write-perm-memory *&optional file-name*) [*function*]

Purpose: Saves all current formulas, their classifications, and properties about them and the constants and variables involved in them in a disk file.

Syntax: *file-name* is optional, and will default to "perm-memory" in the current directory.

Examples:

(write-perm-memory "lrrh-knowledge")

Remarks: Only formulas stored in the main system are saved - information saved in the specialists' representations is not saved yet. Parameter and trace values are not saved either. It is assumed that the identical hierarchy and indicator values will be loaded before the file is read in again. **Write-perm-memory** may be abbreviated to **wpm** .

(read-perm-memory *&optional file-name*) [*function*]

Purpose: Reads in the information saved by a **write-perm-memory** to restore the wffs and constants in the system at that time.

Syntax: *file-name* is optional, and will default to "perm-memory" in the current directory.

Examples:

(read-perm-memory "lrrh-knowledge")

Remarks: This command cannot be executed after a wff has been loaded. The same hierarchy that existed at the time of the writing of the file should already be loaded. **Read-perm-memory** may be abbreviated to **rpm** .

Chapter 2

EPILOG Representation

This chapter describes the logical syntax used for formulas accepted by **EPILOG** , as well as the probabilities the system uses. The logical syntax is intended to be the same as the logical syntax given for episodic logic in Chung Hee Hwang's thesis *A Logical Approach to Narrative Understanding* (see references), although the description here is slightly different (for historic and implementation reasons).

2.1 Logical Syntax Summary

To read this syntax summary, note the following: '*' means 0 or more occurrences, '+' means 1 or more occurrences, '|' and ',' mean choices (as do separate lines for multiline definitions) and{} indicates optionality. Items in *italics* are syntax types, in **bold** are actual input (the **()** are included although they don't really look bold), regular print includes syntax instructions, and comments.

wff->(**(**{negation}*quantifier variable*{*wff*}*wff***)**
(**(**{*negation*}*term pred term****)**
(**(***wff-op wff***)**
(**(**{*negation*}*wff logical-conn wff*+**)**
(**(**{*negation*}*wff episodic-op term***)**
(**(**{*negation*} *wff causal-conn wff***)**
(**(**{*negation*} *wff* **true)**

wff-op -> *name* examples: **nec, poss, probably, perhaps, past, perf, futr, pres, prog**, ...
(**(***sentence-modifier pred***)**
Note:*pred* is a 1 place predicate

sentence-modifier->*name*examples:**adv-s, adv-e, adv-f, adv-p** , ...

quantifier->**A**, **E**, **the**, **most**, **many**, **some**, **few**, **none**
(**(***quantifier-modifier quantifier***)**

quantifier-modifier-> *name*examples:**nearly**, ...

variable->*name*|*name_sort*

sort->**episode**, **ep**, **event**, **set**, **time**, **number**, **num**, **real**, **integer**, **int**, **string**, **propos**

negation->**not**

*logical-conn->***and**, **or**, **implies**, **<=>**, *number-pred*
(*number-pred variable+*) controlled variables

episodic-op-> ******, *****, **@**, **_**

causal-conn-> **because**

name-> lisp symbol name, consisting of a string of characters and numbers,
starting with a character

number-pred-> real number <= 1

term -> constant
(*pred-nominalization-op pred*)
(*sentence-nominalization-op wff*)
(*function term+*)
quasi-quoted-expression
record
quoted-expression

*pred-nominalization-op->name*examples:**K, K1, Ka, To,**...

*sentence-nominalization-op-> name*examples:**that, whether, Ke, YN-q**, ...

*record->***($ ´sort constant+)**

quoted-expression-> quoted list, contents unspecified [1]

*quasi-quoted-expressionR ->***(qquote***wff***)|(qq***wff***)**
(qquote*term***)|(qq***term***)**

constant->name|name_sort| number | string

*function->name*examples:**set-of, date, cardinality-of, start-of, pair, fst, rst,** ...

*pred->name*examples:**kill**, **love**, **eat**, **pretty**, ...
(*pred-modifier pred*)
(*multi-pred-modifier pred+*)
lambda-pred

lambda-pred->lambda-expr
(*lambda-pred term**)

*lambda-expr->***(L***variable wff***)|(L***variable pred***)**

*pred-modifier->number|name*examples:**very, plur, coll, almost, sort-of, former, in-manner, ly** ...
(*modifier-forming-op pred*)

*modifier-forming-op->name*examples:**coll-of, attr, adv-a, nn, na, adv-q**, ...

*multi-pred-modifier-> name*examples:**rel, mos,** ...

In addition, each subpart may be named using the symbol ! and a name just before the closing bracket. These names may then be used instead of the whole expression wherever the expression is legal. For example, *(A x (x wolf ! p1) (x grey) ! p2)*would mean *p2* could now be used for *(Ax (x wolf) (x grey))*, and *p1* could be used for *(x wolf)* - e.g. *(A x p1*

[1] Note that no meta variables (variables over wffs, predicates, etc) may be quantified within a quasi-quoted expression. Any quantification outside a quasi-quoted expression in a meaning postulate or simplification schema must be over a meta variable or a sorted variable.

(x fierce)).

The thesis description of episodic logic has the negation operator acting on a sentence argument, rather than inside the sentence. EPILOG will accept this form as well and move the negation inside.

Notes: Some operators are stored but not used for inference yet. Infix position is used for wffs themselves, prefix for all other constructions. Wherever a syntax type is defined as *type -> name* , the user may add his own names there using the *add-predicate, add-operator,* etc functions. Where there is a fixed set, no new ones may be added (logical connectives, quantifiers, etc). Controlled variables are stored but not currently involved in inference.

This syntax is really somewhat more permissive than intended. For example, the syntax ignores predicate adicity, so it permits [John gives], (very gives), and other oddities. For more details on particular constructions, see the "Logical Syntax Details" section in the User Manual.

Some examples (each followed by the system's attempt at English generation):

*(A x (x wolf) (A y (y human) (A z_ep ((x meet y) * z) ((y in-danger) @ z))))*

If someone is met by a wolf, he is in some danger.

(not wolf1 friendly)

WOLF1 is not friendly.

(nec (A x (x wolf) (x fierce)))

Necessarily wolves are fierce.

((lrrh pretty) and (lrrh friendly))

Little Red Riding Hood is friendly and pretty.

*((wolf1 meet lrrh) ** ep1_episode)*

WOLF1 met Little Red Riding Hood.

*(((lrrh in-danger) @ ep2_ep) because ((wolf1 meet lrrh) * ep1))*

Little Red Riding Hood was in some danger because she was met by WOLF1.

*((wolf1 want (To (L x (E y_ep ((x eat lrrh) * y))))) ** ep3_ep)*

WOLF1 wanted to eat Little Red Riding Hood.

(lrrh (mos pretty girl))

Little Red Riding Hood is the most pretty (prettiest) girl.

((start-of ep2) during ep1)

The start of WOLF1 wanted to eat Little Red Riding Hood while WOLF1 met her.

*((wolf1 ((ly quick) eat) gm) ** ep4_ep)*

WOLF1 quickly ate Grandmother.

2.2 Logical Syntax Details

An informal sketch of the logical syntax is provided here. More details of the logical syntax are given in the documents described under References. Throughout this manual, examples are sometimes shown in the manner described in the article mentioned above, and sometimes in the form that the system accepts, which is slightly different. To convert the form in the paper to the form the system accepts, note the following: () are used instead of [], : are eliminated, $*$ is **, $\forall$ is A, $\exists$ is E, λ is L, $\rightarrow$ is implies, and $\rightarrow_n$ is just n .

2.2.1 Formulas

Formulas are entered in *infix* form, with the first argument preceeding the predicate, and all other arguments following the predicate. This makes it more readable. For example,

(lrrh smaller-than wolf1)

Functions, operators, and other such expressions are always entered in *prefix* form, with the function or operator first, and arguments following. For example,

((start-of e1) before (date 1989 12 01 12 00 00))

2.2.1.1 Quantification

Quantification may be restricted or unrestricted. If restricted, it takes the form of $(Q\alpha\ \Phi\ \Psi)$, where Q is a quantifier, α is a variable, and Φ and Ψ are formulas. Thus, $(\forall\alpha\ \Phi\ \Psi)$ and $(\exists\alpha\ \Phi\ \Psi)$ are equivalent to $(\forall\alpha)\ \Phi \rightarrow \Psi$ and $(\exists\alpha)\ \Phi\ \&\ \Psi$, respectively. For example, *Every bomb is dangerous* is represented as (note that English letters *A* and *E* are used for $\forall$ and $\exists$ for computer implementation and that we use *infix* expression which puts the predicate after its first argument):

(A x (x bomb) (x dangerous))

or

(A x ((x bomb) implies (x dangerous)))

while "There is a hand-made bomb" will be represented as

(E x (x bomb) (x hand-made))

or

(E x ((x bomb) and (x hand-made)))

In addition to the standard quantifiers *A* and *E* , some additional quantifiers are recognized by **EPILOG** . These are *some, many, most, few, WH* and *none* [2] . The current version handles only *A* and *E* completely, although it can make some inferences with the others, and some wh-questions using *WH* can be answered. In addition, some other quantifiers *E!* and *the* are recognized as being similar to *E* , but adding a uniqueness attribute to the specified object. The system currently normalizes top level constructs using these to have an additional part showing the uniqueness, but the exact actions have not been fully determined, so care should be taken when using them.

In existentially quantified facts, the restriction will always get probability 1, and the fractional probability (if this is an inference) will be placed on the main clause.

[2] *none* is actually *no* , except that using *no* causes confusion during verification, as there may be YES and NO atoms temporarily inserted into clauses

Quantification over meta-level objects, such as predicates, is allowed restrictively in schemas - in meaning postulates or simplification schemas. Mp's or simplification schemas must contain quantification over meta-level variables (wffs, predicates, operators, etc) or sorted variables. Additional quantification over over variables must take place within quasi-quoted expressions in these schemas. Regular knowledge rules may not quantify over meta level objects.

2.2.1.2 Connectives

Two types of connectives are used for implication. As shown above, in universally quantified statements, English word *implies* is used instead of traditional implication symbol *rightarrow*. (For semantic reasons, it is preferable to represent *implies* as the probability *1* .) Generic conditionals use implications with lower objective probabilities attached (e.g., *rightarrow*.85), and these implications are expressed by the probabilities themselves. For example, "Most guerrillas are dangerous" could be represented as

((E x (x guerrilla)) 0.85 (x dangerous))

interpreting *Most* as lower objective probability 0.85 (this number just "seems" reasonable. Similar numbers can be chosen for other quantifiers, and the response generator does correspond English quantifiers with numbers, but only for the purpose of producing natural sounding English).

Conjunctive and disjunctive connectives take the form of *and* and *or* respectively (& and | may also be used, and will be converted to *and* and *or* , respectively, by the system). These connectives should be placed *after* their first argument. Note that enumerative connectives can take arbitrarily many (but at least two) arguments. Thus, "There is a male Palestinian guerrilla" may be represented as

(E x ((x guerrilla) and (x male) (x Palestinian)))

Equivalences are represented using the symbol $\Leftrightarrow$ (<=> for Epilog). $(\phi \Leftrightarrow \psi)$ is equivalent to

$((\phi \rightarrow \psi)$ *and* $(\psi \rightarrow \phi))$

2.2.1.3 Episodic and Propositional Operators

Episodic and propositional operators connect formulas to episodes and propositions respectively. Three kinds of episodic operators are used: ** (equivalent to $*$), *, and @. Expressions $(\Phi **\eta)$ and $(\Phi *\eta)$ mean that Φ is an *overall description* of episode η and Φ is a *(partial) description* of episode η, respectively. For example:

*(A x_set (x (coll people)) (A y_ep ((x walk) ** y)*

*(A z (z member-of x) ((z walk) * y))))*

$(\Phi$ @ $\eta)$ is used as an abbreviation for

$(E\ e\ (e\ same-time\ \eta)\ (\Phi\ **\ \eta))$

although the temporal aspect of this does not get passed along to the temporal specialist. It is better to use * and ** and explicitly state the temporal aspect, or add the above as a simplifications schema.

*(A x_wff (A y_term ((qq (x @ y)) true) ((qq (E e_ep (e same-time y) (x ** e))) true)))*

An underscore (_) is used to specify a proposition, *(Φ _ P)* meaning that Φ is the content of proposition P . Then, "John believes that a guerrilla kidnaped Mary" could be represented as

*(E x ((E y (y before now) (E z (z guerrilla) ((z kidnap Mary) ** y))) _ x)*

*(E x1 (now during x1) ((John believe x) ** x1)))*

or, equivalently, as

*(E x ((E y (y before now) (E z (z guerrilla) ((z kidnap Mary) ** y))) _ x))*

and

*(E x1 (now during x1) ((John believe x) ** x1))*

Note that existential variables inside the content of a proposition cannot be freely skolemized. They are however, given unique variable names by the normalizer. In the above formulas, for instance, y and z cannot be skolemized, whereas x and *x1* can.

The following example illustrates some of the syntax discussed so far.

"An explosives-laden car blew up in a Shiite neighborhood in Beirut, police say."

(E p1_propos ((E e1_ep ((e1 before now) and (E x1 ((x1 .SHiite neighborhood))
and
(x1 in Beirut) (e1 occur-in x1))))
(E x2 ((x2 car) and (E e2_ep (e1 during e2)
(E x3 (x3 (plur explosive))
*((x2 laden-with x3) ** e2)))) ((x2 blow-up) ** e1))) _ p1)*
*(E e3_ep (now during e3) (E x4 (x5 police) ((x5 say p1) ** e3))))*

2.2.1.4 Sentential Operators

A number of operators may act on a formula to add information. The **nec** operator makes a formula necessarily true. **adv-p** translates a propositional adverbial - e.g. *((adv-p certain) (John happy))* . **adv-f** is translates a frequency adverbial - e.g. *(past ((adv-f regular) (John see Mary)))* . **adv-e** translates an adverbial over an event - e.g. *((adv-e (in-loc California)) (John see Mary))* .

2.2.1.4.1 Negation

The negation operator is *not* (~ may also be used, and will be converted to *not* by the system). To minimize the levels of nesting of clauses, the negation operator is moved inside its argument expression. Thus, "John is not a rebel" will be formulated as

(not John rebel)

If a formula is entered with the negation outside its argument expression (e.g. *(not (John rebel))*), the system will move the negation to its preferred location.

2.2.2 Predicates

So far the predicates illustrated have been simple entities (names), but more complex predicates can also be handled. In particular, lambda abstracts can be used to represent predicates with arguments, and predicate modifiers can modify a predicate in various ways.

2.2.2.1 Lambda Abstracts

Lambda abstracts express more complex predicates than can be handled by a single symbol, although they can also express those. They can be used in predicate position as a predicate, or more commonly, be used in a construction with an operator that requires a predicate as its argument (especially nominalization operators - see below). The English letter *L* is used for λ. For example, predicates *happy* and *kiss Mary* can be represented as

*(L x (L y ((x happy) ** y)))*

and

*(L x (L y ((x kiss Mary) ** y)))*

Lambda reduction and conversion is done during the normalization process for lambda expressions used in predicate position, where the subject is a symbol. For example,

*(John (L x (E y ((x happy) * y))))*

would be converted to

*(E y ((John happy) * y))*

and this wff would replace the original one. Similarly,

*(John ((L x (L y (E z ((x love y) * z)))) Mary))*

would be converted to

*(E z ((Mary love John) * z))*

2.2.2.2 Predicate Modifiers

A number of predicate modifiers are recognized (and you can add more if you desire - see the section on adding operators). For example, *coll* or *plur* modify a type predicate to mean a collection or group of entities of that type. To express that *c1 is a pack of wolves* , *(c1 (coll wolf))* can be used. Some other modifiers are intensifiers, such as *very* or *extremely* . These operate on appropriate predicates (not types!) - for example, *(lrrh (very pretty))* . (Notice that predicate modifiers such as *coll* or *very* are expressed in *prefix* form.) There are also non-monotonic operators, such as *sort-of* , and *almost* . The predicate they modify is not true of the entity it is being predicated of (*(couch1 (almost red))* does not mean that *couch1* is *red*).

When used in inference, a modified predicate will not match the unmodified version of the predicate - meaning postulates should be used to make any such valid inferences. For example, something that is *very* pretty is also pretty, while something that is *almost* pretty is not pretty.

2.2.3 Terms

Terms are the arguments to the predicates and functions used in formulas in **EPILOG** . So far only simple named constants and variables have been shown, some of them with sort information attached. Additional types of terms can also be represented, including other kinds of constants (numbers, strings, and more complex entities), and complex terms involving operators. This section describes these other kinds of terms, as well as some caveats on naming terms.

2.2.3.1 Sorts

Sorts are a subset of the main (fixed) type hierarchy. They differentiate arguments so that specialists can tell whether or not something is in their domain. Currently the sorts are: *episode, time, set, propos, string, number, integer,* and *real* .

There are two ways to have a sort attached to an item - one is to do it manually, by following the argument name with an underscore and the sort (e.g. *e1_episode*). (This is the way **ECoNet** appended sorts, so to maintain some sense of consistency between the two implementations, the same method is used here.) Note that this is the same underscore used as the propositional predicate _. Since there are no spaces around the underscore when used as a sort attacher, no confusion should result. Currently, no testing is done to ensure that the sort given is a valid one. Abbreviations are allowed for the longer sorts. Recognized abbreviations are in *()* : *episode (ep, event), integer (int), string (str)* , and *number (num)* .

The other method is to allow the system to automatically put it there for you, which it can do in certain restricted circumstances. If a constant is asserted to be of that type (for example, *(e1 episode)*), the sort will automatically be added. Also, arguments in the final position in a proposition with predicate **, * or @ will have sort *episode* automatically added to them, although if they occur in a quantified expression with a restriction, it is usually a good idea to put the *_episode* on the quantified variable anyway - if the restriction is complex the sort sometimes isn't put on in time to be useful. Similarly, the final argument for propositions with predicate _ will have sort *propos* added. In some complex formulas, allowing the system to automatically add the sort may result in the sort being added too late for some operations, so it is a good idea to manually add sorts when quantifying variables, or when adding a new constant in a long, complex sentence. Another possibility is to explicity add the sort as a type restriction. For example, *(A x_episode (A y ...))* could also be input as *(A x (x episode) (A y ...))* .

Once a constant has been given a sort, it never loses it, so after the first use (e.g. *n1_number*), the item may be referred to simply by its name (*n1*), without losing sort information. Variables have this property also, within a single proposition. The sorts on variables are wiped out before the next propostion is entered.

2.2.3.2 Constants

Symbolic names, simple numbers and strings may simply be used as themselves. Examples:

(mm_number less-than 3)

(('time 1988 12 01 00 00 00) before e1_episode)

(city1 has-name "Edmonton")

2.2.3.3 Records

For complex entities (such as dates or sets), we have introduced records, which are similar to the quoted expressions **ECoNet** used. A record is a list of the form *($ 'sort ...)* , in argument position in a proposition, where *sort* is a legitimate sort, and the rest of the list may contain almost anything. An example of a date is *($ 'time 1988 12 01 00 00 00)* . Note that substitution and function evaluation within a record are handled.

2.2.3.4 Quoted Expressions

A future enhancement will be to handle arbitrary list structures and symbolic expressions, including real "quoted expressions" of the form $'(\tau_1\ \tau_2\ ...\ \tau_n)$. Currently these may be input to the system, but their use is limited. No substitution or function evaluation occurs within a quoted expression.

2.2.3.5 Quasi-Quoted Expressions

Quasi-quoted expressions are list structures which are similar to quoted expressions, except that limited substitution may be done into them. They look like *(qq ...)* or *(qquote ...)* . Variables over meta-level objects (wffs, predicates, operators, etc) may be substituted for, and meta-level functions may be evaluated. No other substitution or evaluation takes place inside a quasi-quoted expression though. Quantification is allowed inside these expressions, but not over meta-level variables. These expressions are used mainly in meaning postulates and simplification schemas where complex formulas need to be matched to make a rule fire, or where the resulting inference is quantified or complex. Quasi-quoted expressions most often occur with the *true* predicate. For examples, note the quasi-quoted expressions in the following meaning postulates:

(A x_pred (A y_term ((qq (y (very x))) true) ((qq (y x)) true)))

(A x_wff (A y_wff ((qq (most z x y)) true) ((qq ((E z x) 0.8 y)) true)))

There is more discussion of this in the section of meaning postulate and simplification schema syntax.

2.2.3.6 Functions

Functional terms are lists which consist of a function name (no operators allowed!) followed by a number of terms which are arguments to the functions. A few have been predefined, and some of the specialists recognize some functions (e.g. *set-of* , *start-of* , etc).

2.2.3.7 Naming Conventions

When a formula is entered, the formula and each subpart are normalized to atoms which contain all the information about the portion - if there is an operator, the arguments, probability, etc. The system can generate names for the atoms, or the user may name them by putting *! name* before the closing parenthesis. For example, *(A x (x wolf) (x fierce) ! wolfrule)* would be normalized, and the resulting atom would be named *wolfrule* . The names, user and system generated, are available to the user to use in other formulas where the identical subpart is desired. (The "long" form of any name is available on it as its *'print* property).

Normalization also standardizes variable names, and skolemizes if possible. The standardization of variable name ensures that they will have all the necessary information with them whenever they are used - thier sorts, and whether or not they are matchable variables. Matchable variables are those that can be unified with a constant, leading to an input driven inference or a goal reduction. These are variables which occur in a positive context (for example, a rule consequent or fact) with quantifier **A** , or those which occur in a negative context (for example, a rule antecedent) with quantifier **E** . For example, in

(A x (x wolf) (E y (y girl) (x friend-of y)))

x is matchable, and y is not. In

((E x (x wolf)) 0.8 (x grey))

x is matchable. For goal chaining, the reverse is true, with the matchable variables occurring in a positive context, and the non-matchable ones being in a negative context. For example, x in *(E x (x wolf))* is matchable if this is a goal (for backward chaining).

Standard variable names are *x, y, z, x1, y1, z1, ...* for matchable variables, and *u, v, w, u1, v1, w1, ...* for non-matchable variables. Variables with sorts will be normalized to these names as well, with a short form of the sort and a dash in front (e.g. *ep-x, set-y, num-u, int-v, real-x1, time-z*). Although almost anything can be used as a variable name on input, these names should NOT be used for constants. Also, a "sorted" variable (e.g. *ep-x*) should not be used to represent a variable of a different sort. Lambda variables are not matchable (except with other lambda variables), and have L- in front of the standardized name.

When an existential variable has been skolemized, the new constant will have the name c followed by some number (e.g. *c3*), or a sort, followed by a dash, the c and a number (e.g. *ep-c14*). It is not recommended that the user enter any of his/her own constants using that form, and especially one should not use that form for variables.

2.2.3.8 Nominalization Operators

The nominalization operator *To* takes a predicate as its argument, and the whole construct becomes a term. This is where the lambda abstracted form of predicates comes in quite handy. For example, "John wants to kiss Mary" can be represented as

*(E e1 (now during e1) ((John want (To (L x (L y ((x kiss Mary) ** y))))) ** e1))*

As seen in this example, lambda abstracts are particularly useful when one wants to use a construct containing a predicate *and* some argument(s) as a predicate.

In addition to *To* , there are kind forming operators *K* and *K1* , as in *(K (plur rebel))* , "rebels" as a kind, or *(K1 bomb)* , "a bomb" as a kind. For example,

((K wolf) fierce)

represents "Wolves are fierce". Note that meaning postulates must be used to convert these to appropriate forms for inference (depending on the predicate, a formula involving such an operator can be interpreted in different ways - e.g. "Dogs are friendly," "Dogs are barking," "Dogs are widespread" will all have different interpretations, although they can all be expressed using the kind operator).

2.2.4 Operators

Operators can modify wffs, predicates, quantifiers, terms, and even other operators! A number of them have already been discussed under predicate modifiers and nominalization operators. A number of oper-

ators are recognized by the system, including the sentential operators *nec* and *poss* , predicate modifiers *very, almost, sort-of, plur* and *coll* , nominalization operators *K, K1,* and *To* , and the adverb forming operator *ly* .

2.2.5 Meaning Postulate and Simplification Schema Syntax

Meaning postulates (MPs), i.e., axiom schemas expressing analytical facts about various operators and types of predicates (etc), can be used to generate useful input-driven inferences, in a manner entirely analogous to the use of ordinary axioms. In particular, MPs in the form of conditionals are accessible for input-chaining via one or two top- level operators in the antecedent. Simplification schemas are axiom schemas expressing equivalent forms for wffs, and can be used to replace an input wff by its (more useful) equivalent. They MUST express an equivalence!

Both types of axiom schemas are input using the same syntax as the rest of the system (using command **mp**), except that quantification over formulas, predicates, operators, and quantifiers is required. Additional sorts (*pred, wff, operator, quantifier, term* , etc) are used. Quantification over non-meta level entities is only allowed for sorted variables (i.e. over episodes, sets, etc). Quantification over other non meta-entities must be done inside quasi-quoted expressions. Variables may only be used in place of wffs, predicates, operators, etc inside a quasi-quoted expression.

*(A x_wff (A y_term (x ** y) (x * y)))*
is a meaning postulate stating that "characterizing descriptions" of episodes are also partial descriptions, (this is just an example - currently the system recognizes this without explicitly storing the inference).

(A x_pred (A y_operator (y monotonic) (A z (z (y x)) (z x))))
states that a monotonic operator (like *very*) modifies a predicate without changing its truth value (currently this "meta" information about predicate, operators, etc, can be entered just like story information - *(meta '(very monotonic))* - and a *meta-specialist* is also used to help get the information when necessary).

For some mp's, it is necessary to match on larger units than simple literals. For example, if we want an mp to apply to formulas with a particular quantifier, say *most* ,

(A x_wff (A y_wff (most z x y) ...))
will have *x* and *y* as its trigger keys - that is not what we want! (and also has quantification over a non meta-entity - z). In such cases, quasi-quoted expressions should be used, in conjunction with the predicate *true* . The mp,

(A x_wff (A y_wff ((qq (Most z x y)) true) ((qq ((E z x) 0.85 y)) true)))
forces the system to match the entire *(Most z x y)* part to fire the rule. *((qq wff) true)* is equivalent to *wff* and so can be used whenever desired. The difference between using the two different representations is the trigger keys that result from them - the quasi-quoted representation gives larger trigger keys. In the above example, note that the normalization would preserve the fact that the two *z* 's are the same (because they are in quoted formulas). Normally these would be normalized to two different variables.

If the above rule were instead to be used as a simplification schema,

(A x_wff (A y_wff (((qq (Most z x y)) true) <=> ((qq ((E z x) 0.85 y)) true))))
any formula of the form *(Most x ...)* would be REPLACED by the equivalent form *((E x ..) 0.85 ...)* .

2.3 Adding New Syntactic Entities

2.3.1 Adding New Predicates

New predicates may be defined by adding them to a type or part hierarchy (which makes them type predicates), adding them to predicate hierarchy, adding topic indicators to the predicate names, or adding a new specialist which handles the predicates. However, the system will also "assume" that a new object it finds where it expects a predicate is a predicate.

(add-predicate *&key type part hier subnodes parent hier-type indicators part-type package entry-rtn*) [*function*]

Purpose: adds a new predicate.

Syntax: If *type* is *t* , this will be a type predicate. To be a type predicate it must be on a type hierarchy. You may specify the hierarchy name with *hier* , parent node with *parent* and subnodes with *subnodes* , and even give the hierarchy type with *hier-type* . If they are not specified, a new hierarchy will be invented for the predicate automatically. It is usually easier to add type hierarchies using the **add-hier** command (described a little later in the manual), but this serves to add type predicates quickly and easily for testing. For type predicates which are also sorts, *sort* should be set to the sort so that automatic sort addition can be done to entities declared to be of that type. Part predicates are specified when *part* is t. A small hierarchy is built for these predicates as is done for the type predicates. Part-type predicates (exist on both a part and a type hierarchy) may be specified by setting *part-type* to *t* , although the system usually can calculate and set this itself.

Topic *indicators* may be added here, or through the command **add-indicate** . *Specialists* is a list of specialists interested in the predicate. The specialists are usually added during activation of a specialist, so this will rarely be needed by a user.

For externally defined routines to be accessed through the "other" specialist, a *package* to find the evaluation routine should be specified. If there is in addition an entry routine to save information using this predicate, *entry-rtn* should also be specified.

Examples:

```
(add-predicate 'pillow :type t)
```

simply adds a type predicate *pillow* , without adding any connections for it (other than to *entity*). This is useful for on the fly testing when there isn't time to bother with deciding where something should go on which hierarchy.

```
(add-predicate 'next-to :package 'space :entry-rtn 'space::set-next)
```

would add the predicate *next-to* , which would be evaluated and stored by an external routine in the *space* package.

2.3.2 Adding New Functions

Functions must be added *before* they are used. **EPILOG** will not "guess" that they are functions.

(add-function *fnname &key rel-pred sort specialists package*) [*function*]

Purpose: Adds a new function.

Syntax: *fnname* . The keys are all optional. If this function may be evaluated by an external routine (through the "other" specialist), the package the other routine lives in should be specified in *package* . *Specialists* is a list of the specialists that are interested in the function. *Rel-pred* is the predicate this function can be transformed to if necessary during flattening for specialists. *Sort* indicates the sort of the object resulting when the function is applied.

Examples:

(add-function 'time-of)

just adds *time-of* as a function. This is probably all that most users will need, as specialists usually add the functions themselves.

(add-function 'nearest-neighbors :package 'space)

adds the function *nearest-neighbors* , which would be evaluated using the external routine *space::nearest-neighbors* .

2.3.3 Adding New Operators

A user may desire more operators, and this section describes how to add them. To allow the system to interpret the new operators correctly as operators and to know what kinds of arguments should follow an operator, some additional commands have been set up. New operators MUST be defined as operators - the system cannot "guess" that they are operators.

To add operators, the following commands will be useful:

(add-operator *operator &key op-type result-type arg-types indicators specialists*) [*function*]

Syntax: *operator* is the name of the new operator, and either the type of operator *op-type* , or the resulting type *result-type* and the types of the arguments *arg-types* must be specified. Types which may be used are *pred* , *operator* , *wff* , and *term* (all as defined in the syntax summary), as well as any predefined or user defined operator types. Predifined operator types are: *pred-op* (makes a *pred* out of a *pred*), *term-op* (makes a *term* out of a *term* , and *wff-op* , makes a *wff* out of a *wff*). *Indicators* is a list of topic indicators for this operator. Unless these are specified, the operator will generally be ignored in the classification phase of the system. This works well for operators like *very* , but operators like *make* are probably important enough that they should have special classifications (so *indicators* should be specified for them). *Specialists* is a list of interested specialists - unless you are adding a new specialist do not include this field!

Examples:

(add-operator 'nec :op-type 'wff-op)

adds the necessity operator, which is a *wff-op* , where *wff-op* is a predefined operator type with result *wff and argument type wff* .

(add-operator 'K :result-type 'term :arg-types 'pred)

adds the kind operator *K* , which takes a *predicate* and makes it into a *term* .

Remarks: This does not allow the definition of operators which might take more than one different type of argument, or which require several arguments of different types. In addition, the number of arguments cannot be specified, so if too many are given the syntax checker will not be able to warn the user. Predefined operator definitions are in the file *epi/epilib/objects.lisp* .

(add-operator-type *op-type &key result-type arg-types*) [*function*]

Purpose: adds a new operator type.

Syntax: Operators defined to be of this type create objects of type *result-type* , and accept arguments of type *arg-types* . This can be used to simplify definitions for operators if there are a large number which take the same type of arguments and return the same type of result.

Examples:

(add-operator-type 'wff-op :result-type 'wff :arg-types 'wff)

2.3.4 Adding New Sorts

New specialists may find that inventing a new sort for their domain is useful. To do so:

(add-sort *sort &key nicknames*) [*function*]

Syntax: *nicknames* is a list of other possible names for *sort* . These are usually shorter than *sort* . The first nickname is taken as the short form for the sort and will be used in building variable and constant names for entities of that sort.

Examples:

(add-sort 'episode :nicknames '(ep event))

2.3.5 Adding New Quantifiers

Although most of the quantifiers imaginable are already recognized by the system, it is possible that there are more one might want to add. These must be added before using them. Note - the average user will never need to use this! Most of the quantifiers one could ever need are already there!

(add-quantifier *quant &key prob negation distributive*) [*function*]

Purpose: adds a new quantifier *quant* .

Syntax: If applied as part of a rule, *prob* is included in the probability calculations of the result. If *prob* is not specified, 1 is used. *Negation* is a lambda expression denoting how to negate an expression involving the quantifier. *Distributive* is *t* if the quantifier is distributive.

Examples:

(add-quantifier 'most :prob 0.8)

(add-quantifier 'WH :prob 1)

2.4 Probabilities

Each proposition has a probability associated with it that is the lower subjective probability of that proposition. Input formulas will all have probability 1, but inferred formulas will have a probability that is a combination of the probabilities of all the formulas used to make the inference, and the rule's probability. For example, if we infer *(lrrh pretty)* from the input fact

(lrrh girl)
and the input rule

((E x (x girl)) 0.7 (x pretty))
(lrrh pretty) will have probability 0.7.

If it is desirable that an input formula have a probability less than 1, it should be input as *'(YES number wff)* . This will be normalized to *wff* with probability *number* .

If an existentially quantified formula is inferred, the probability of the restriction will be 1, and the probability of the main clause will be based on the rules and wffs used to infer it. This is because restrictions generally contain type and time relationship information which is considered "presuppositional". For example,

*((E x (x girl)) 0.7 (E y (y building) (E z_ep ((x live-in y) ** z))))*
applied to

(lrrh girl)
would yield

(c1 building)
with probability 1, and

*((lrrh live-in c1) ** ep-c2)*
with probability 0.7.

Sometimes the same formula (or its negation) can be inferred using several different paths. In such cases we would like to combine the various pieces of evidence and recalculate the probability of the formula. Currently the system can combine several answers during the question answering phase, and determine the resulting answer and probability, but this has not been extrapolated to input driven inference yet. There are still some problems with ordering of the evidence, as well as using the combined probabilities later on in future calculations. These should be resolved soon.

In addition to the numeric probability stored with each formula, a support-set is also associated with it, which is a list of the formulas whose probabilities need to be taken into account when inferring a new formula from this one. For an input formula, the support-set is the formula's probability. If a formula is split into several wffs, a special probability atom is created to hold the probability before the split, and this atom is in the support-sets of the new formulas. Otherwise the support-set consists of given or conjunctive wff-names, and objective probabilities used in inferring the proposition.

This ensures that we do not include the same subjective probability of a proposition more than once in a calculation (although the objective probabilities should be included for as many times as the rule is actually applied).

One of the recent improvements to the system was to allow it to combine several pieces of supporting or contradictory evidence. There are some problems still in determining exactly what should and shouldn't be combined, which may lead to supporting combinations where only one should be selected (probabilities a bit high). The probability produced in combination seems accurate, but the support set information is not currently being dealt with properly. This means that later uses of the combined formula may result in inaccurate probabilities (too low). So far our examples have not required much adjudication so the magnitude of the discrepencies is not known. We are currently working on this.

Chapter 3

Assertions and Other Input

To get information into the system, it must be asserted. There are several kinds of information the system works with. Formulas use the syntax described in an earlier chapter, and may be story facts, general knowledge rules, meaning postulate axiom schemas, simplification schemas, or facts about specific predicates and other meta-level objects.

Other input to the system, in particular the hierarchies, must be input using special commands. Someday the system may be smart enough to automatically build the hierarchies based on input rules, but for now they must be built manually.

3.1 Types, Parts and Topics

3.1.1 Hierarchies

EPILOG needs to know some information about the predicates used in formulas in order to topically classify them properly. Predicates are broadly divided into type and non-type predicates, where types are treated specially by classification (both the main and topical classifications). A type predicate is any predicate that exists on a type hierarchy. These predicates are monadic - they take one argument, Predicates which should be placed on type hierarchies are basic kinds - nouns like *thing, wolf, human, chair* , etc. Monadic predicates which should NOT be on type hierarchies include adjectives such as *pretty* , *happy* , *red* , etc.

In addition to providing the type distinction, the hierarchies are also "climbed" in order to find other applicable rules for a fact (e.g. girls are humans, so any rule applying to humans also applies to girls), or descended to find more specific facts or individuals for a rule (e.g. if we are looking for a creature, a wolf will do nicely). The hierarchies are also used by the *type-specialist* (to be discussed later) to quickly determine relations between type predicates.

A fixed type hierarchy is automatically loaded when the system starts up, and contains the sort hierarchy, as well as an entity hierarchy. In addition to these, a set of optional type hierarchies is included, in the file *eg.hier* . New hierarchies or extra entries for the given hierarchies are easily added using the **add-hier** command.

The type hierarchies used by **EPILOG** are similar to those used by *ECoNet* , with some extensions. Each hierarchy has a name, a root, and some properties associated with it. The root and hierarchy name are usually the same, but need not be, especially if it is desirable to have the same root broken

down in several different ways (giving different hierarchies). Any number of type hierarchies may exist at one time. Each is treated independently, unless there are identical predicates in more than one. These "connections" can then be used to determine relationships between predicates on different hierarchies when possible. A particular type predicate may be broken down in any number of ways, with the root node being the same and a different hierarchy name for each breakdown.

Another property of the hierarchies indicates whether a type hierarchy or a non-type predicate hierarchy - type hierarchy is the default. Predicates on these other hierarchies have no special properties (unlike the type predicates) other than fast determination of relations between them by virtue of being on a hierarchy. They are useful when a number of predicates are special cases of some other predicate - for example, there are a number of different actions that could be considered "repairing".

The type hierarchies are used both for organization of input facts and rules (the type hierarchies in particular), and for fast determination of relations between predicates on them. A preorder numbering scheme is used to number the nodes on the hierarchy so that relationships between nodes in the same hierarchy can be determined in constant time. The numbering is done in a depth first manner, with each node containing an id number of its own, and the maximum id of any subnode beneath it. A comparison between these numbers indicates whether a subsumption or disjointedness relation holds between nodes. One property of the hierarchies decides the relation between sibling nodes. On an *exclusion* hierarchy, all sibling nodes considered disjoing, and so are incompatible. In **overlap** hierarchies, the relation between sibling nodes is unknown. Inference using rules must be used to determine the relation between sibling nodes. Only subsumption relations may be determined on such hierarchies. All hierarchies are considered **exclusion** unless the function **set-hier-type** is used to indicate otherwise.

Part hierarchies also exist in the system. Each level of the hierarchy indicates a breakdown into component parts. Tangling of part and type hierarchies is allowed and seems to give the intuitively correct responses by the system. These are much like type hierarchies - parts are also types - but are not "climbed". The part specialist also uses these hierarchies. The usefulness of these hierarchies is currently limited by the fact that we cannot yet represent exhaustiveness in the hierarchies, so we cannot say that "Little Red Riding Hood does not have a tail" based soley on information in the hierarchies.

Consistency testing is optionally done on entries to the hierarchies and can detect loops and other irritations (set ***specialist-entry-effort*** to something greater than 0). Full consistency testing for tangled hierarchies os co-NP-complete, but the checks done here will catch most errors. This is strongly recommended for new hierarchies and testing. For hierarchies that are used on an everyday basis and not changed often, you should tweak the flag to 0 so that the hierarchy will load faster, but remember to tweak it back after!

Hierarchies may be exhaustive or non-exhaustive. Part hierarchies are assumed to be exhaustive unless otherwise indicated; type hierarchies are assumed non-exhaustive unless otherwise indicated. Exhaustiveness is implemented at a hierarchy level, not on a node by node basis, so if a particular node in an exhaustive hierarchy is not exhaustively subdivided, a remainder subnode should be added for it. A remainder subnode is assumed to be one which ends in *-remainder, -other* , or *-rest* . Currently exhaustiveness on part hierarchies is used in conjunction with a set of familiar parts to determine when a particular type of part cannot belong to a particular type of entity. Exhaustiveness on type hierarchies is currently not being used.

Sample type hierarchies and a sample topic hierarchy are located in file *eg.hier* in the *test-files* directory.

(add-hier *hier-name parent-node child-nodes*) [*function*]

Purpose: Creates and adds to type/topic/predicate hierarchies.

Syntax: Adds to hierarchy named *hier-name* . If this hierarchy did not previously exist, *parent-node* is made the root. Each node in *child-nodes* is added under *parent-node* .

Examples:

(add-hier 'thing 'creature 'animal 'human)

(add-hier 'explode 'explode 'blow-up 'go-boom)

Remarks: If a non-type predicate hierarchy is desired, use **(set-hier-type** *hier-name* **'pred-hier)** before any **add-hier** commands.If **specialist-entry-effort** is set > 0, the new entries are checked for consistency before adding to the hierarchy. The hierarchy name and root node may be the same, but hierarchy names must be unique, while root nodes do not have to be.

(set-hier-type *hier-name type*) [*function*]

Purpose: Indicates various hierarchy properties, including whether hierarchy is exclusion or overlap, exhaustive or non-exhaustive, part hierarchy types (form, function, state) or whether the hierarchy is a non-type hierarchy (pred-hier). By default, all hierarchies are exclusion; part hierarchies are exhaustive and type hierarchies are non-exhaustive; part hierarchies are of type form.

Syntax: *type* is one of **overlap, exclusion, exhaustive, non-exhaustive, form, function, state** , or **pred-hier** . *Form, function* , and *state* are elements on the tweakable parameter ***part-hierarchy-types*** . If the user wishes, he may add to this list. The only requirement is that some consistency be maintained in adding the types to the part hierarchies.

Examples:

(set-hier-type 'occupation 'overlap)

(set-hier-type 'explode 'pred-hier)

(set-hier-type 'animal-parts 'non-exhaustive)

(set-hier-type 'human-parts2 'function)

(add-part-hier *hier-name parent-node child-nodes-with-bounds*) [*function*]

Purpose: Creates and adds to part hierarchies.

Syntax: Each child node must be in the form *(node bound)* or *node* . If no bound is given, exactly 1 is assumed. Bounds should be in the form *number* (for exactly that number of parts), or *(min max)* , where *max* may be nil if unlimited. If the node is a remainder node (i.e. ends in *-other, -remainder, -rest*), bounds of 0 to infinity will be automatically added for it if none are specified by the user.

Examples:

(add-part-hier 'human-parts 'body '(leg 2) '(arm 2) 'trunk)

(add-part-hier 'human-parts 'mouth 'tongue '(tooth (0 32)))

To print a hierarchy, use the **display** command with topic *hier* , and the names of the hierarchies to be printed (e.g. *(display 'hier 'entity '-f)*). If you specify *-full* or *-f* , the entire hierarchy will be

displayed, recursively; otherwise only its properties and connections will be shown. If no hierarchies are specified, they all will be printed.

3.1.2 Topic Indicators

One part of the classification process requires "topics". Topics are considered to be those items which exist on a special hierarchy called *tp.topics* . During topical classification, predicates are expected to supply a list of topics to use. To do this, predicates contain a list of indicators, which consist of either a topic, or a topic and particular argument positions (so the classification of a formula may have different topics for each argument in the formula). For example, the predicate *eat* indicates topic *tp.feeding* for the *subject* argument (the eater), and *tp.existence* for the *object* argument (the thing being eaten). This causes

(wolf1 eat lrrh)

to get classified under *(wolf1 tp.feeding)* , and *(lrrh tp.existence)* . Classification will be discussed more later. The predicate *before* has an indicator of *tp.time-relationship* , with no argument numbers supplied, so all arguments will be classified with the same topic. If no indicators are associated with a predicate, a warning will be printed, and the default topic *tp.unknown* will be used for all arguments.

The topic hierarchy contains some topics which the system expects to be there. These special entries are:

tp.type - types of individuals classified here

tp.spec - instances of types classified here

tp.content - formulas involving _ classified here

tp.description - episodic formulas involving **, * and @ classified here

tp.mental-attitude - modal subnets classified here

tp.role - parts and roles classified here

tp.mental-attitude-object - modal propositions about facts

tp.unknown - new predicates with no information are stored here

To add additional topics (never remove the given ones!), use the **add-hier** command described in the previous section. (The file *eg.hier* also contains a sample topic hierarchy).

The rest of this section describes commands which can be used to add the topic indicators to predicates. The file *eg.indicate* in directory *test-files* contains a sample of indicator links.

(add-topic *topic predicate-list*) [*function*]

Purpose: Adds indicator links to predicates for a topic

Syntax: *predicate-list* consists of a list of predicates, optionally followed with a role. An indicator link to *topic* is added to each predicate in *predicate-list* .

Examples:

(add-topic 'tp.emotional-giver 'love 'subject 'kiss 'subject)

Remarks: (Note - previously this worked with argument numbers instead of roles - this is still supported). Indicators may be added to operators in the same way.

(add-indicate *predicate indicator-list*) [*function*]

Purpose: Adds indicator links for topics to a predicate

Syntax: Each entry in *indicator-list* is either a topic, or a list consisting of a topic and an argument role. An indicator link is added to *predicate* for each entry in *indicator-list* .

Examples"

(add-indicate 'try '(tp.experimenter subject) '(tp.experiment object))

Remarks: (Note - previously this worked with argument numbers instead of roles - this is still supported). Indicators may be added to operators in the same way.

To display the topic indicators, and other information about a predicate, use the **display** command with topic *'pred* - for example *(display 'pred 'eat)* . The *-full* and *-brief* options have no effect.

3.2 The Assertion Process

When a formula is entered, negations are distributed, top level conjuncts split, and skolemization is done for top level existentially quantified variables. The resulting wff (or wffs if it has been split) is (are) then normalized. After normalization, simplification schemas may operate on the formula, replacing it by an equivalent form, which then goes through the entire normalization procedure again. Then the resulting formula is tested for consistency and simplified, and finally stored under the appropriate classifications. (Thus, when a formula - fact or rule - is loaded, one may notice that it is in a slightly different form from the one given by the user.) Story sentences are also stored in a special array ***input-array*** which will eventually be used for narrative inference.

For story facts and knowledge rules, **EPILOG** makes input-driven inferences from world knowledge, and semantic discourse knowledge. World knowledge may take the form of either a universal implication or a generic conditional. Semantic knowledge consists of meaning postulates, i.e. necessary universal axioms or axiom schemas. (For input-driven inference, the necessity operator can be ignored). Meaning postulates are always applied when appropriate, but inference with other kinds of knowledge may be controlled by a number of parameters. These parameters control whether or not forward inference is even attemted (***story-forward*** , and ***rule-forward***), as well as when an inference chain is terminated (***interest-threshold*** , etc). These parameters and the processes they affect will be described in more detail shortly.

3.2.1 Asserting Formulas

Use **knowledge** to load general knowledge, such as rules, **story** to load specific story facts, **goal-knowledge** to load rules which should be used only during question answering, **meaning-postulate** to load meaning postulate axiom schemas, **simplification-schema** to load simplification schemas, and **meta** to load assertions about meta-level objects, such as predicates.

The formulas loaded should all conform to the logical syntax described in Chapter 2. Formulas entered using **knowledge** , **story** or **goal-knowledge** may not contain quantification over meta-level entities, such as predicates, or have predicates or operators as terms. Formulas entered using **meta** may not contain quantification over meta level entities either, but may make assertions about predicates and operators where they are used as terms. Formulas entered using **meaning-postulate** or **simplification-schema** MUST contain quantification over meta-level variables. Any other quantification must be contained within a quasi-quoted expression. Use of variables in predicate, wff, or operator position must be done within quasi-quoted expressions as well.

Load all hierarchy and topic indicator information **before** story and world knowledge. Additional hierarchy and indicator information may be added at any time, but the system will not reclassify what is already there. Any predicates should be either on a type hierarchy or have indicator links associated with them before being used - if these are not present, the system will print a warning, and assume the topic *tp.unknown* .

When asserting story facts, always assert the type of an entity first. This will ensure that applicable rules all fire when subsequent data is entered.

(knowledge *&rest formulas*) [*function*]

Purpose: Loads linguistic or world knowledge.

Examples:

(knowledge '(A x (x wolf) (x predator)) '(A x (x child) (x person)))

Remarks: *Knowledge* can be abbreviated to *kn*.

(goal-knowledge *&rest formulas*) [*function*]

Purpose: Loads linguistic or world knowledge which is to be used only for goal chaining, not for input driven inference.

Examples:

(goal-knowledge '(A x (x human) (E y (y man) (y father-of x))))

Remarks: **Goal-knowledge** can be abbreviated to **goal-kn** or **gkn** . The only difference between **goal-kn** and **kn** is that formulae entered through $goal - kn$ are not used during input-driven inference.

(story *&rest formulas*) [*function*]

Purpose: Loads story sentences.

Examples:

(story '(LRRH girl) '(E x1 (x1 wolf) (x1 bad)))

Remarks: The system performs input-driven inferences as story sentences are loaded. To see inferences that are being made, make sure you are tracing *forward* (or *entry*). Input driven inferencing

continues until the inference made is not interesting (product of probability and interest is less than ***interest-threshold***).

The only difference between *story* and *kn* is that formulae entered through *story* are also kept in an input array. This array is not currently used for anything except display purposes, but could possibly be used in inference merging later.

(meaning-postulate *&rest formulas*) [*function*]

Purpose: Loads schematic meaning postulates.

Examples:

(mp '(A x_pred (A y_term ((qq (y (almost x))) true) ((qq (not y x)) true))) '(A x_wff (A y_wff (A z_term ((qq ((x and y) * z)) true) ((qq ((x * z) and (y * z))) true)))))

Remarks: $Meaning-postulate$ can be abbreviated to *mp*.

(simplification-schema *&rest formulas*) [*function*]

Purpose: Loads simplification schemas.

Examples:

(simp-schema '(A x_wff (A y_wff (((qq (most z x y)) true) <=> ((qq ((E z x) 0.85 y)) true)))))

Remarks: $Simplification-schema$ can be abbreviated to $simp-schema$ or *ss*. The difference between *ss* and *mp* is that *ss* inferences REPLACE the original formula, where *mp* inferences are added in addition to the original formula. *SS* formulas are also required to be equivalences.

(meta *&rest formulas*) [*function*]

Purpose: Loads meta information to be used during meaning postulate inference.

Examples:

(meta '(kill action-pred))

Remarks: This is similar to *story* except that additional flexibility is allowed in the sysntax (i.e. having predicates as arguments), and no forward inferencing is done.

3.2.2 Re-asserting Formulas

Sometimes it is interesting to reassert a formula to see inferences can be made, especially if some new information has just been added.

(reassert *phi-name*) [*function*]

Purpose: To reassert a given formula so that input-driven inferencing can be tried again for it.

Examples:

(reassert 'wff1)

Remarks: The formula is not stored again, in either the knowledge base or specialist domains, nor is classification recomputed. If the formula is not found, nil is returned. Only input-driven inferences are affected. This is useful when new rules have been added after story information (and ***rule-forward*** is not on), or when type information for an entity is added after other information. For a single entity, this can be used in the following manner:

(mapcar 'reassert (retrieve 'entity))

where retrieve can get all information (as it would here), or only a specific topic, or to redo the entire knowledge base:

(mapcar 'reassert (get-everything))

3.3 Normalization

When a wff is asserted, inferred, or questioned, it first goes through a normalization process. This process transforms the given list form of the formula into an atom which contains all the information as properties. Each subpart is also an atom containing all its information as properties. In addition, operator distribution is done, skolemization is done if applicable, variable names are standardized, sorts are stripped off terms, and the wff is split into several wffs if applicable (for conjunctive wffs and conjunctive episodic wffs). For example,

((lrrh girl) and (lrrh pretty))

would be split into separate wffs

(lrrh girl)

(lrrh pretty)

and

*((lrrh meet wolf1) and (wolf1 meet lrrh)) * e1)*

would be split into

*((lrrh meet wolf1) * e1)*

*((wolf1 meet lrrh) * e1)*

(the latter case can now be handled by meaning postulates, but currently is still being handled by the normalization phase).

In addition, simplification is done to eliminate any YES or NO atoms in the formula. Formulas with patterns *((qq wff) true)* or *((that wff) true)* are normalized to *wff* (at the top level). This enables more complex meaning postulates to be written, and the resulting inference will be normalized to the expected result.

During the normalization process, the formula and each subpart are made into atoms. These atoms are kept in a hash table, hashing on the longer list version of the item. Whenever we normalize a new item, we check to see if we have done that subpart before, and if so, can immediately return the atom corresponding to it. If not, a new atom is created, and the relevant properties added. The hash table is checkpointed and retracted with the rest of the system.

The parts of a disjunction or conjunction, at whatever level of embedding they occur, are always ordered in a consistent manner to facilitate looking them up however they are specified. This means that *((wolf1 grey) or (wolf1 black))* will be identical to *((wolf1 black) or (wolf1 grey))* .

A note about skolemization: Since an existentially quantified variable can occur outside the scope of the quantifier, the user should keep track of the variable names introduced with an existential quantifier. This problem will disappear once a logical translator is written.

Top level episodic formulas with an embedded conjunction are currently split (e.g . *(((lrrh smile) and (lrrh happy)) ** e1)* is split into *((lrrh smile) * e1)* and *((lrrh happy) * e1)*). For the average user this is desirable (and so the default is to split), but the original formula is lost, and in some instances this is not desirable. A tweakable flag controls whether or not this is done - ***split-episodic*** . If this flag is turned off, the splitting is not done automatically. A meaning postulate

*(A x_wff (A y_wff (A z_ep ((qq ((x and y) * z)) true) ((qq ((x * z) and (y * z))) true))))*

should be added to do it. Then you'll have the original and split formulas.

The normalization process also checks to ensure that all quantified variables in meaning postulate and simplification schemas are meta-variables - i.e. have a meta sort on them (*term, wff, pred* , etc). In addition, simplification schemas must be equivalences.

3.3.1 Controlling Normalization

Trace values which show the normalization process:

lambda - shows lambda conversion

norm - shows normalization

Tweakable parameters which can affect normalization:

stop-if-error
: indicates whether the system should pause and wait for the user to press return when an error is detected in the syntax of an input formula. The default for this is t - stop and wait. If lisp is not being run interactively, the system will determine this and set the flag off when it starts up.

quoted-indicators
: a list of symbols which indicate that what follows is a quoted list for EPILOG. This is initially set to be just the lisp quote symbol *(')* .

name-symbol
: the symbol which indicates that the next item following is to be used as a name for the preceeding structure. Initially this is set to *!* .

expand-names
indicates whether or not named symbols (using the symbol above) should be expanded when printed. This is initially set to t. Note, if you change this, it will not affect the printing of formulas already input, only new ones.

split-episodic
indicates whether or not top level episodic formulas with embedded conjuntions should be automatically split. The default is t - split them. If this is turned off, a meaning postulate should be entered to make the split inference, as follows

*(A x_wff (A y_wff (A z_ep ('((x and y) * z) true) ('((x * z) and (y * z)) true))))*

check-pred-parts
when a new predicate is input which can be split into several parts (between -),this flag indicates that both parts must exist as predicates on their own to make this a "compound" predicate, in which case it will inherit properties from the last part. Otherwise only the last part is checked (the default - nil)

The application of simplification schemas uses exactly the same inference procedure as the meaning postulate inference (to be described shortly) with the exception that the inference REPLACES the original formula rather than augments the knowledge base. This means exactly one simplification schema can operate on a formula. Simplification schemas are applied recursively to the resulting formula, until no more are applicable. No schema can apply more than once.

Although the simplification schemas are equivalences, they are applied in one direction only - as if they were implications. Their usual role is to transform a less preferable input form into a more preferable input form. If a less preferable form is entered, it will be transformed into the more preferable one, but we don't want the reverse taking place! When using them, expect them to be applied like implications, but make sure that they are entered as equivalences or the system will complain.

3.3.2 Controlling Simplification Schema Application

Trace values which show when simplification schemas replace a formula:

simplification-schema - shows original and new unnormalized form

Note that application of these schemas can be traced by tracing the same values that are available for watching meaning postulate inference.

3.4 Classification

Formulas are classified to figure out where to store them so that they are most accessible for later use. The classification is also used as a starting point to find rules, meaning postulates and other facts which might be applicable to a given formula. Two different classifications are used for story facts and general rules: one which is based on the arguments of a formula, and the predicate itself (the main classification), and another based on the arguments and a topic suggested by the predicate (the topical classification). Both classifications are required, as the main classification gives the fastest access for inference, and the topical classification groups the formulas together in a logical manner for display and partial answers.

For example, if someone asks "Does John love Mary?", in the absence of other information, we might answer "Well, he admires her." Meaning postulates are classified a little differently and will be described shortly.

Classification uses simple lists as indices. The main classification is of the form:

(concept predicate role)

or

(concept specialist-name)

Topical classification is of the form:

(concept topic)

The parts of the classification lists will be described in more detail shortly. For example, *(lrrh pretty)* would be classified under

(lrrh pretty subject)

for the main classification, and under

(lrrh tp.appearance)

for the topical classification.

*((wolf1 eat lrrh) ** e1)* would be classified under

(lrrh eat object)

(wolf1 eat subject)

(e1 episode-specialist)

for the main classification, and

(wolf1 tp.feeding)

(lrrh tp.existence)

(e1 tp.description)

for the topical classification.

These classification lists are used to index into a hash table. Each entry in the hash table is either a list of proposition names, or another hash table (for subnets). Subnets will be described later.

The classification phase has several parts to it - first type information for variables is temporarily saved, keys and trigger keys are determined, and then the main classification is calculated. This classification is then used to get the topical classification as well.

Type predicates are used with predicates to classify rules, so that rules which have no chance of succeeding because of type clashes are not tested (there is no point in looking at a rule which applies to foxes meeting rabbits when Little Red Riding Hood meets the wolf!). If an rule argument (variable) has no type available, *entity* is used. If the input is translated from English, we should always have a type available, since it is usually included in the sentence. For example: "anyTHING that is in someTHING is smaller than it" or "If someTHING is in someTHING, it is smaller than it". In EPILOG this would be

*(A x ((x thing) and (E y (y thing) (E z ((x in y) * z)))) (x smaller-than y))*

or

*((E x (x thing) (E y (y thing) (E z ((x in y) * z)))) implies (x smaller-than y))*

and would be classified under

(thing smaller-than subject)

(thing smaller-than object)

and also topically classified under

(thing tp.size-to)

(thing tp.size-from)

Type predications in **EPILOG** are classified under

(individual type-specialist)

for individual constants, as well as under *(pred)* for positive type predications. For example, *(wolf1 wolf)* would be classified and stored under *(wolf1 type-specialist)* and *(wolf)* . This allows the system to find all instances of a particular type easily.

Functions are classified as if they were formulas - using the function name (or specialist name) and the arguments. For example, *((cardinality-of set-c1) greater-than 3)* will get classified under

(set-c1 set-specialist)

Most operators are treated as modifiers, and so are ignored for the classification. However, operators with indicators attached are treated as important, and are used like predicates in the classification.

Within each classification, wffs are further organized in order of increasing complexity, and decreasing probability. Complexity is calculated based on the number of literals, and the type of arguments (constants or variables). This is initally set so that each literal has a base complexity, to which is added the complexity of its arguments (where constants are considered to have higher complexity than variables). This calculation may be changed by tweaking the parameters ***literal-complexity*** , ***variable-complexity*** , ***constant-complexity*** , and ***function-complexity*** .

There are no topic access skeletons (which were used in **ECoNet**) - instead,if you wish to know everything about a particular individual, you must hash on each pair consisting of that individual and a topic (using the topically organized table instead of the main one) - the routine **retrieve** (or **display-concept**) does this for you. The topic access skeletons no longer serve a useful function (except in that one case), and do take up a lot of storage. **(display** *individual*) will give the topically indexed facts about an individual.

3.4.1 Key Selection

Before the classification itself is done, the trigger keys for the formula are determined. For an input fact, the trigger key is the fact itself; for a conditional statement, there will be at least two trigger keys (one from the antecedent, one from the consequent), and these are what must be matched to trigger the rule for both input-driven or goal-directed reasoning. Only the trigger keys will be classified, and the classification of these keys is the classification used for the whole formula.

First the positive (consequent) and negative (antecedent) keys of the rule are determined - this is done so that each variable is represented at least once, but no more often than necessary, and episodic keys are preferred to non-episodic keys. These lists are then checked to find the trigger keys. The trigger keys are determined based on their interest values - the most interesting key from the negative keys is chosen, and the most interesting from the positive keys. If several keys are "tied" for most interesting, or if their interest is above a threshold value (***key-threshold***), they will all be trigger keys.

For example, the rule "Any physical-object which is in a location when a bomb explodes near that location is probably destroyed"

((E x1 (x1 location)
(E x2 (x2 physical-object)
*(E e1 ((x2 in x1) ** e1)*
(E e2 ((e2 during e1) and (e2 occurred-near x1))
*(E x3 (x3 bomb) ((x3 explode) ** e2))))))*
0.9 (E e3 ((e3 episode) and (e2 cause-of e3))
*((x2 destroyed) ** e3)))*

has many positive and negative keys. The negative keys (which make the rule fire on input story facts) are:

*(x1 location), (x2 physical-object), ((x2 in x1) ** e1), (e2 during e1),*
(e2 occurred-near x1), (x3 bomb) , and *((x3 explode) ** e2).*

Clearly we don't want this rule to fire every time it encounters a physical-object, or a temporal relation, or a location! The negative keys selected here as appropriate and most interesting are: *(e2 occurred-near x1)* and *((x3 explode) ** e2)* . Both of these keys are required because there is no way to guarantee order of input. These will become the "trigger" keys, and the rule will be classified according to the classifications of those keys. Type information is extracted from the rest of the rule first to fill in the classification. Similarly, the positive "trigger" keys will be *(e2 cause-of e3)* and *((x2 destroyed) ** e3)* .

Selection of keys can be changed by modifying the interest levels of predicates, and also by changing the key interest threshold (***key-threshold***). Any key which has an interest level above this threshold will automatically be included, regardless of other considerations.

If the ***forward-full*** flag is tweaked on, the ***key-threshold*** flag MUST be tweaked to 0. Otherwise only specific keys will be selected as triggers, and since partial instantiations are not allowed, some desirable forward inferences may not be made.

Formulas involving the predicate *true* have as keys the first argument, which is usually a quasi-quoted formula. The quasi-quoted formula is itself the key - it is not split into smaller literals.

3.4.2 Main Classification

Classification of formulas is done to store new ones, and to look up existing ones which can be compared against a given formula, either for input driven or goal directed inference. Formulas are classified using lists of the following form:

(concept predicate role)

or

(concept specialist-name)

where a *concept* is either an individual constant or a type predicate, and *role* is one of *subject, object, iobj* . If the predicate has specialists associated with it, the second form is used; otherwise the first form is. This allows that formula to be retrieved and then compared with other formulas that particular specialist is interested in (because specialists allow formulas with different predicates to be compared). If the predicate is a non-type predicate, and is on a hierarchy, the root of the hierarchy is used instead of

the predicate itself (or the specialist). These hierarchies are generally not tangled, and define a class of predicates (like different types of *repair*). The roles are automatically assigned by the system, as follows: The first argument is always has role *subject* . If there are exactly two arguments, the second one is *object* , otherwise the second one is *iobj* and all others are *object.*

General knowledge will usually be classfied under (type-pred predicate subject) indices, while specific story information will usually go under *(individual predicate subject)* indices. For example,

(lrrh small)
will be classified under

(lrrh small subject)
and

*(A x (x wolf) (A y (y girl) (A z_ep ((x meet y) ** z) ((y in-danger) @ z))))*
would be classified under

(wolf meet subject)

(girl meet object)

(girl in-danger subject)

Where operators on predicates are involved, the assumption is made that most are just modifiers, and so the operator arguments are more important. So *(lrrh (very pretty))* will be classified under *(lrrh pretty subject)* . However, this is not true in all cases. The system uses the presence of topic indicators on an operator to tell it that the operator is important. It then combines the operator arguments and the arguments to the modified predicate to determine the classification. So *(john (make juice))* will get classified under *(john make subject)* and *(juice make object)* .

It is important when writing rules to ensure that the types of the arguments are included, since as the above example shows, they are essential to the classification of the rule. If the types are not included, "entity" will be assumed. Rules are classified using both the antecedent and the consequent of the rule, since they may be applied in either the forward or backward direction.

3.4.3 Topical Classification

In addition to the main classification, which is required for inferencing, a topical classification is done, to use for displaying information, and perhaps later to assist in answering some kinds of wh-questions (such as *What does Little Red Riding Hood look like?*). The main classification itself is used to generate the topical classification. Topical classifications are of the form:

(concept topic)
where a *concept* is either an individual constant or a type predicate. The concept part comes directly from the main classifiation, and the predicate and role (or specialist name) are used to determine the topic. General knowledge will usually be classfied under *(type-pred topic)* indices, while specific story information will usually go under *(individual topic)* indices. For the examples given earlier,

(lrrh pretty)
will be classified under

(lrrh tp.appearance)
and

*(A x (x wolf) (A y (y girl) (A z_ep ((x meet y) ** z) ((y in-danger) @ z))))*

would be classified under

(wolf tp.social)

(girl tp.social)

(girl tp.existence)

3.4.4 Modal Classification

In addition to the simple list classifications, there is another classification method for peoples' "mental worlds". Each individual has his/her own subnet(s), where all formulas about what he or she believes, thinks, wishes, etc are kept. For the main classification, there is a subnet for each possible modal predicate, while in the topical classification, all beliefs, hopes, etc are kept together under the topic *tp.mental-attitude* . (Tweakable parameters ***modal-topics*** and ***subnet-topics*** contain this information. Although they are changable, doing so may be dangerous.) Within such a subnet may be other subnets corresponding to formulas about what a person believes/thinks/wishes another person believes/thinks/wishes. For mental world information in **EPILOG** , classification is as follows: Main classification:

((concept1 modal-pred subject) (concept2 modal-pred subject) ... (concept pred role))

Topical classification:

((concept1 tp.mental-attitude) (concept2 tp.mental-attitude) ... (concept topic))

concept1 , *concept2* , ... are either individual constants or type predicates, *modal-pred* is one of *believe, hope, wish, etc* , and the last classification is the classification of the proposition that the other concepts have the attitude about. For example, *John believes Joe said Mary kissed Bill.* would be classified under:

john believe subject

joe said subject

mary kiss subject

and

john believe subject

joe said subject

bill kiss object

and topically under:

john tp.mental-attitude

joe tp.mental-attitude

mary tp.emotional-giver

and

john tp.mental-attitude

joe tp.mental-attitude

bill tp.emotional-object

The first pair is used in the main hash-table to get the subnet's hash table, and then the next is used to index into that hash table, and so on.

3.4.5 Part/Role Classification

One method of handling classification of parts is to put all parts of an individual into a separate subnet. This appeared promising early on, but as the mechanism to handle parts is still undergoing revision, and a part specialist has been included in the picture, it is still under evaluation. Until resolved, parts are not being entered into separate subnets.

However, we can have specific part types. Part hierarchies are actually orthogonal to the type hierarchy. Each node in the type hierarchy has a part hierarchy associated with it, but we are not representing those as yet, as we cannot capture the inheritance. What it amounts to is that for each part, there is a hierarchy that mirrors the type hierarchy. For example, in the type hierarchy we have *creature -> animal human* , and *human -> adult minor* , etc. For **head** , we have a corresponding hierarchy *creature-head -> animal-head human-head* , and *human-head -> adult-head minor-head* , etc. Then when we index rules about heads, we can look under the appropriate one. We would then like to have rules in the knowledge base to the effect

((E x (x person) (E y (y head) and (y part-of x)))) implies (y person-head))

rather than explicitly typing the head each time. However, in practise, for rules, we need to include that type so that things are indexed properly. For example, if we wanted a rule that said *anyone with a green head is likely ugly* , we would have to phrase it as

((E x (x person) (E y ((y person-head) and (y part-of x)) (y green))) 0.8 (x ugly))

so that the rule is indexed also under *(person-head green subject)* . Then later if we find an instance of a person's head being green, we can find this rule again. This aspect of classification is still undergoing revisions.

3.4.6 Meaning Postulate and Simplification Schema Classification

Since the axiom schemas used for meaning postulates and simplification schemas are so powerful, we must be very careful to only use them in restricted circumstances. During input-driven inferencing, they are applied in the forward direction only, and during goal-driven inference, only in the backward direction. To manage this, a special form of classification is used for them, which is also a list, but its contents differ somewhat. The second entry of the list is the top level operator, and the first is either the next level operator, or the type of object the first one is being applied to. Note that if a quasi-quoted expression is involved, the contents of it are used to determine the classification. Both antecedent and consequent keys are used, as, like other rules, meaning postulates may be used in both directions. So

(A x_pred (A y_term ((qq (y (very x))) true) ((qq (y x)) true)))

would get classified under *(pred very)* and *(term pred)* , while

(A x_pred (A y_operator (y nonmonotonic) (A z_term ((qq (z (y x))) true) ((qq (not z x)) true))))

would get classified under *(pred operator)* , *(operator nonmonotonic)* , and *(term pred)* .

*(A x_wff (A y_term ((qq (x ** y)) true) ((qq (x * y)) true)))*

would be classified under *(wff episode-specialist)* (since both * and ** are handled by the episode specialist), while

*(A x_wff (A y_wff (A z_term ((qq ((x and y) * z)) true) ((qq ((x * z) and (y * z))) true))))*

would be classified under *(and episode-specialist)* and *(* and)* .

Input formulae are classified in this special way to find applicable meaning postulates to apply, but are not stored under these classifications. Both the antecedents and consequents are classified, although

the mp's may only be applied in the forward direction during input-driven inference, and in the backward direction during goal-driven inference. Also note that there is no topical classification done for mp's.

3.4.7 Controlling Classification and Storage

The parameters which may be tweaked to affect classification and storage are as follows:

default-indicators
A list of indicators to use for new predicates which do not specify indicators of their own. This is initially set to *(tp.unknown)* , but may be changed to something more meaningful to your own system if you will be entering a large number of predicates which belong to a particular category.

key-threshold
The threshold of interest beyond which all keys will be included in the list of trigger keys for a rule. These trigger keys are then used to classify the rule, as well as for matching against input formulas.

literal-complexity
The base complexity for each literal in the complexity calculation. This is initially set to 2, but may be changed using the *tweak* command.

variable-complexity
The complexity for a variable in the complexity calculation. This is initially set to 0, but may be changed using the *tweak* command.

constant-complexity
The complexity for a constant (or record, or quoted expression) in the complexity calculation. This is initially set to 1, but may be changed using the *tweak* command.

function-complexity
The complexity for a functional term in the complexity calculation. This is initially set to 2, but may be changed using the *tweak* command.

subnet-topics
Topics that indicate separate subnet storage (*tp.mental-attitude and tp.role*). Note: it may be dangerous to change this.

useful-nonepisodic-topics
a list of topics which are considered "important" enough to ensure that literals involving predicates which indicate these topics should be included in the trigger keys for rules. This is initially set to *(tp.causal-relationship tp.kinship tp.happening tp.location)* .

input-array-expansion-size
indicates the increment by which the input array is expanded. This is initially set to 100. Changing it should not be necessary unless it is known that a huge number of story sentences are to be input.

The trace values which can be used to watch classification:

class - traces start and results of classification

entry-class - traces classifications used to store each wff input to or inferred by the system.

keys - traces selection of trigger keys for rules

memory - traces expansion of input array

3.5 Consistency Testing and Simplification

Upon assertion, the input formula is tested to see if it is already true, or contradicts known information. How hard it tries to determine this can be decided by the user - it can just do a simple lookup or evaluation, a simple backchaining attempt, or a full blown proof by contradiction. (The default is lookup and specialist evaluation)

If the formula already exists, or a contradiction is found, adjudication is necessary to combine the different pieces of evidence. If the new formula contradicts previous information, the higher probability will be adjusted up, and the lower down. If it supports previous information, the probability will be adjusted up. All descendants of formulas whose probabilities have been adjusted are recalculated. However, further use of these formulas may then have inaccurate probability calculations. The system also has trouble determining whether or not the two items should be combined - if one inference path subsumes the other, or is more specific, the two should not be combined, but the system cannot detect most cases of this yet.

Currently the default mode is to attempt the adjudication. The system can be told to operate like a previous version by changing the ***consistency-action*** flag to *1* - then it will accept contradictions, but warn the user, and print that a duplicate is found if supporting wffs are detected. Also, if a new supporting formula has higher probability than the old one, it is accepted, and the probability becomes the higher one.

Consistency testing is also done on some other kinds of input, including additions to the hierarchies.

The user may indicate the effort the system is to use in determining consistency of input with the parameter ***consistency-effort*** . Legal values of this parameter are:

0 - no consistency testing
1 - lookup only
2 - verification using only unit probability propositions (including specialists)
3 - verification using any applicable propositions (including specialists)
7 - full blown question answering

If the consistency effort flag is 1 or 2, this is also used to simplify the input clause. Anything other than that requires an additional verification to simplify. Actually in this case both should be entered (if we verify using something other than probability 1, that constitutes an inference), but currently only one is entered.

Consistency testing is also done on entries to the hierarchies - this was discussed in the section on Hierarchies earlier in this chapter.

Given that the system has detected an inconsistency, the user may also indicate the action to be taken using ***consistency-action*** . Legal values for this are:

0 - enter anyway
1 - print a warning, and then enter anyway
2 - reject the input clause
3 - print a warning, and then reject the input clause
4 - attempt to combine the evidence using adjudication

The default currently is 4, although it can be reset to 1 to act like previous versions of the system. If the input supports previous evidence, its probability will be adjusted up; if it contradicts, the higher probability will be adjusted up and the lower one down. All descendants of formulas whose probabilities have been adjusted have their own probabilities recalculated (and their descendents, and so on). There are two main problems with the adjudication right now. The first is that further calculations using an adjudicated formula will not take into consideration formulas that were used in inferring it, so their probabilities may be multiplied in again - this can lead to probabilities lower than they should be. The other is that combination will sometimes take place when it shouldn't because the inference paths are equivalent, or one is more specific than another and should be used instead of it. This can lead to probabilities too high. These and other related issues are currently under investigation.

3.5.1 Combining Supporting or Contradictory Evidence

For formulas where several pieces of supporting and/or contradictory evidence have been combined, the probability support sets of each piece of evidence are kept with the formula, on a supporting list and a contradictory list. Each time a new piece of evidence is incorporated or the probability is to be recalculated for some other reason, all the support sets are used to recalculate the probability. First all the supporting evidence is combined, using the probability calculation

$$(1 - (1 - p_1)(1 - p_2) \ldots (1 - p_n))$$

where the p_i are the calculated probabilities from the support sets of the supporting evidence. The same calculation combines all the contradictory evidence. Then there is one supporting probability and one negating probability, and these are combined and recalculated as follows:

$$p_s = (p_s - (p_s * p_c)) / (1 - (p_s * p_c))$$

and

$$p_c = (p_c - (p_s * p_c)) / (1 - (p_s * p_c))$$

where p_s is the probability resulting from combining all the supporting evidence, and p_c is the probability resulting from combining all the contradictory evidence. The calculation using both contraditory and supporting evidence is order dependent, so doing it in this way helps to avoid problems with that.

3.5.2 Controlling Consistency Testing

Tweakable parameters which can affect consistency testing:

consistency-effort
: Indicates how hard to try to determine if an assertion is inconsistent with already known facts. Levels are 0 (no consistency testing), 1 (lookup only), 2 (verification with unit probabilities only), 3 (verification with anything), and 7 (full blown question attempt).

consistency-action
: Indicates what to do if inconsistent assertions are made. Actions are 0 (enter anyway), 1 (print a warning and enter anyway), 2 (reject), 3 (print a warning and reject), 4 (try to combine).

Once a wff is asserted, whether directly by the user or through an inference, input driven inference is attempted on it to determine the consequences of that information. Any consequences are then asserted to the system, and undergo the same procedure of normalization, classification, consistency testing, and input-driven inference. This stops when there are no more rules to apply, or the resulting inference is

not acceptible either because it is not very "interesting" (more on this later), or because its probability is not high enough. Meaning postulates are always applied however, regardless of interest or probability. Meaning postulates are applied whenever applicable, while knowledge rules may be applied only once in an input driven inference chain. Various controls are available on the input-driven inference mechanism - see the details section to find out more if you are interested.

3.5.3 Meaning Postulate Inference

The first thing done with the new formula is to classify it as if it were a meaning postulate (but not to store it there). This classification is then used to find mp's in the knowledge base which might be applicable. The input wff is compared to the keys of any mp found, and if it compares successfully, the system tries to match the rest of the keys in the mp, and to make an inference using it. Once the original classifications have been exhausted, the system goes up a level (by finding the parents of both parts of the classification), and repeats the procedure. So for a fact like

(lrrh (very pretty))

the system would first look under the classification

(pred very)

and then try

(pred operator)

where it would find the rule

(A x_pred (A y_operator (y monotonic) (A z_term ((qq (z (y x))) true) ((qq (z x)) true))))

and, given that *very* is known to be monotonic, it would infer

(lrrh pretty)

Previous versions of **EPILOG** had schematic meaning postulates built into the system as inference rules rather than allowing them to be entered as semantic or world knowledge by the user. In fact, the meaning postulates

$(\forall\ \eta\ (\Phi\ *\ \eta)(\Phi\ *\ \eta))$

$(\forall\ \eta\ ((\Phi\ and\ \Psi)\ *\ \eta)((\Phi\ *\ \eta)\ and\ (\Psi\ *\ \eta)))$

are still built into the current version of **EPILOG** as inference rules (the first inference isn't made explicitly, but is made whenever necessary during comparisons). Currently the second one is actually built in as a simplification schema (the split formula REPLACES the original), but turning the ***split-episodic*** flag off can stop that. The system can now accept meaning postulates (using the command **mp**), and can normalize and classify some of them.

The meaning postulate inference is similar to that using regular rules (next section), except that it can only work in the forward direction, and quantification over formulas, predicates and operators is allowed (resulting in unification and substitution of such creatures during the inference process).

When a formula is entered, it is classified as if it were a meaning postulate, but not stored there, and this classification gives the system the places to look for possible applicable meaning postulates. Comparison of keys and evaluation of the resulting formulas can be assisted by the *meta-specialist* , which is described in a later chapter. Basically this specialist allows literals about meta information to be evaluated, using recursively defined predicates, if necessary (like *action-formula*).

Unlike input-driven inference with regular rules, which terminate when an "uninteresting" fact is inferred, mp's are always applied to an input clause, no matter what its interest level.

3.5.4 Controlling Meaning Postulate Inference

Trace values which can be used to watch meaning postulate inference:

mp-eval - traces evaluation of literals during meaning postulate inferencing

mp-test - shows comparisons between keys during meaning postulate inference

mp-access - shows accesses of formulas during meaning postulate inference

mp-class - shows classifications checked during meaning postulate inference

mp-all - shows everything during meaning postulate inference

mp-int - shows interesting parts of meaning postulate inference (everything except classification)

mp-min - shows the inferences made, and evaluations during meaning postulate inferencing

3.5.5 Input Driven Inference Machinery

The classification the input formula was actually classified under is used to obtain rules (or facts) from the knowledge base to apply. A similar process to that of Meaning Postulate inference is used, although keys both in the antecedent (for regular forward inference) and in the antecedent (for contrapositive inferences) may be matched. Once a key is matched, the system tries to match the other keys from facts in the knowledge base. Keys which are not matched are discarded, and when all keys have been matched, and all the substitutions and simplifications have been done, an inference is made.

Like the meaning postulate mechanism, once the original classifications have been exhausted, the system goes up a level, but only using the first part of the classification pair (for subnet classifications, it goes up the first part of each pair in the list, yielding a number of different combinations). For example, for

(lrrh pretty)

the system would first look for applicable rules under the classification

(lrrh pretty subject)

then under

(girl pretty subject)

then under

(child pretty subject)

and so on. In doing so, if it found a rule like

((E x ((x human) and (x pretty))) 0.8 (x happy))

it would infer

(lrrh happy)

with probability 0.8. If it also found a rule like

((E x ((x human) and (x sullen))) 0.9 (not x pretty))

it would make a contrapositive inference of

(not lrrh sullen)

with probability 0.9.

Special stops are put in so that the same rule is not applied more than once in a forward inference chain (to prevent computing transitive closure), and to prevent contrapositive inferences from being made with a rule whose consequent is a type predication (e.g. if the rule states that *anything that eats is a living thing* , we don't want inferences like *the rock never eats*).

The system does *partial rule instantiation* , where not all the keys need to be matched to make an inference. This can result in inferences which are rules themselves, and this may not be desirable for some applications. To avoid this, tweak the parameter ***forward-full*** to *t* , and the system will demand that all variables be matched for a rule to be applied, and that the result not be conditional. If you choose to use this however, you should also set ***key-threshold*** to 0, or you will have an ordering problem with rule applications.

This used to be called "Forward Inference Machinery", since the only inferences made on input were in the "forward" direction - if the antecedent of a rule could be verified, the consequent could be inferred. For example, if we know all wolves are grey, and we have a particular wolf, we can infer that he is grey. Now the system can do this inference in both directions - forward, and backward (contrapositive). So if the consequent of a rule can be disproven, you can infer the negation of the antecedent. For example, if we know any fierce wolf is mean, and we know that a particular wolf is not mean, we can infer that he isn't fierce (if he was, we would infer that he was mean, and that isn't true!).

Here is a more complicated example of input-driven inferencing. Consider world knowledge "If a dog sees a cat, it will probably chase it" and story fragment "Fido is a dog; Morris is a cat; Fido sees Morris," represented in logical form as follows:

R1. *((∃ x (x dog) (∃ y (y cat) (∃ e1 ((x see y)∗ e1))))*
→. 8 (∃ e2 ((e2 right-after e1) & (e1 cause-of e2)) ((x chase y)∗ e2)))

S1. *(Fido dog)*

S2. *(Morris cat)*

S3. *(now during EP1)* ; after

S4. *((Fido see Morris)∗ EP1)* ; skolemization

Notice that rule R1 will be stored under *(dog see subject)* and *(cat see object)* among others. S1 and S2 cannot trigger rule R1 as they do not contain any predicates which indicate *tp.perception* . When S4 is loaded, however, the system looks under *(Fido see subject) (Morris see object) (dog see subject) (cat see object) (animal see subject) ...* , for rules to be tried for input-driven inference, and hence finds R1. Once S4 is successfully unified with a part of R1, namely, ((x see y)∗ e1), with bindings {Fido/x, Morris/y, EP1/e1}, **EPILOG** next checks whether the rest of the antecedent with the same bindings is known to be true. *Fido* and *Morris* must have been classed as type *dog* and *cat* respectively when S1 and S2 were loaded, and thus the consequent of R1 would be concluded by instantiating the variables with the same bindings. Thus, the program infers

(∃ e2 ((e2 right-after EP1) & (EP1 cause-of e2))
((Fido chase Morris)∗ e2))

with lower subjective probability .8 (the probability is obtained by multiplying the lower subjective probabilities of formulas – default is 1 – that verify the antecedent of the rule and the objective probability of the conditional itself). This method of making inferences was earlier called *rule instantiation* . It can be considered a generalization of *modus ponens* , universal instantiation, and "statistical *modus ponens*

". (It is also closely related to Andrew's "general matings" and Bibel's "connection method".)

The general method for input-driven inferencing is as follows. During the standardization of variables, literals which occur positively and negatively are also determined, and saved on the wff. For a rule, the positive literals are usually in the consequent, and the negative ones in the antecedent. The triggering (most interesting) keys of the positive and negative literals are selected and saved. Wffs are then retrieved from the knowledge base, based on the classification of those keys. Each wff is checked to see if it is of the appropriate type to use. Currently if a rule or disjunctive fact is entered, a non-conditional, non-disjunctive fact may be applied to it; and vice-versa. This may not be general enough, but requires more investigation (disjunction management in general is not handled yet).

When a wff is obtained to attempt for possible application to a given input, the trigger keys are searched to find one which resolves with the input fact. The matching key literal is replaced in the rule by NO for non-negated keys, YES for negated ones. The resulting substitutions are made throughout the rule and in the remaining keys as well. We then remove the key from the set of keys, and also remove any keys made up entirely of constants. The remaining keys are ordered with those with the least number of variables first. Next, we try to find all formulas which resolve with those keys. When all keys are matched (or we can't do anymore), we simplify the result, using formulae whose probability is 1, and specialist evaluations.

This process is called "rule instantiation". If all keys are matched (i.e. all variables bound), we have *total* rule instantiation; otherwise *partial* rule instantiation. Sometimes inferences made from the latter are of questionable usefulness. To force full instantiation, the parameter **forward-full** may be tweaked to t , but if this is done, **key-threshold** should also be tweaked to 0.

Forward inference continues to try to infer new formulas from the input clause and the resulting inferred formulas until the following condition is met:

> The inference is not very "interesting". The combination of probability and interestingness is not greater that ***interest-threshold*** , and the inherited interest is not above 0 either. Interestingness will be discussed shortly.

Even when the stopping criteria are met, the current wff is still loaded - but no more input-driven inferencing will be attempted on it. The justification is that after all that work making the inference, we might as well keep it, even if it isn't particularly interesting or high in probability.

Note that the same rule is not applied more than once - this prevents producing the transitive closure of an axiom, and makes things faster. It also prevents infinite looping for recursively related axioms (e.g. there is person, so there is a building that person lives in, which was made by a person, who also lives in a building ...) Also, contrapositive inferences where the consequent key matched is a type predication are not done (to prevent rules like "Anything that eats is a living thing" from generating inferences like "rock1 never eats").

Type information should be entered before other knowledge about an entity. This problem is due to the way rules are classified - usually under the types of the entities. To get around this ordering problem, all facts about an entity would have to be reasserted any time new type information is given about it. This could be very slow, so is not done automatically. A user can use the command **reassert** to get the same effect though.

In normal operation of the system, rules should be given before story facts to be used in input-driven inference, although this is not always necessary. For example, given "Fido loves every one of Mary's cats; Morris is Mary's cat" in the specified order, the program infers "Fido loves Mary"; however, if they are entered in reverse order, the same inference will still be made, if Fido and Mary are still

salient. The distinction between rules and story information is quite blurred (consider "LRRH likes the cake in the basket" "LRRH likes everything in the basket" "LRRH likes everything her mother bakes"). Our tendency is towards eliminating the distinction altogether, and the current input-driven inference mechanism reflects that. There are some controls on the input-driven inferencing: the flag ***rule-forward*** , which is initially set to *t* . If it is tweaked to *nil* , no story facts will be retrieved to apply a new rule to. In addition, forward inferencing for story facts is controlled by the flag ***story-forward*** , which is initially on. If this is turned off, forward inferencing will not be attempted at all.

Note: Disjunctions in rules are now handled properly, and specialists may be consistently used in determining whether a key matches a fact. Fact-fact inference with disjunctions is now handled as well. For example, if we are given that the wolf is either nasty or happy, and we later find out that the wolf is not happy, we can now infer that the wolf is nasty. And if a rule key is *(not x gt 1)* , we can use the number specialist to match it with *(n1 gt= 3)* .

Forward inferencing is not attempted on disjunctive facts right now - this will have to wait for some sort of "disjunction management".

3.5.6 Inference Termination Criteria

As mentioned before, the criteria for determining whether or not to continue making inferences given a particular inference is based on how "interesting" that inference is. The interestingness termination test is to prevent huge numbers of useless and uninteresting inferences from being made when a large body of rules is present (such things as "lrrh is a girl" generating "lrrh has a head", "lrrh has an eye", ..., "lrrh has a mother", "her mother has a head", ...).

Interestingness, in this system, is a numeric value given to a concept or a formula which indicates in a quantitative way how "interesting" or "valuable" a piece of information is. This is used to decide what facts are worth following up and which aren't. For example, *the wolf ate Little Red Riding Hood* is far more interesting than *the wolf ate a cookie* , and we want to follow up the consequences of the first fact, but not necessarily the second. The interestingness depends on the individuals involved, what is being predicated about them, how improbable that is, and possibly other factors as well. Not all of these factors are used in the current version; the interestingness is currently based on the "interestingness" of the argumentsin a formula, and the "interestingness" of the predicate involved. In addition, a formula may "inherit" some interest from the formula which it was inferred from. This "inherited interest" can overcome some dips in the interestingness in chains of input-driven inferencing.

An individual's interestingness is the sum of the interestingness contributions from all the formulas the individual is involved in. So the more we know about something, the more interesting it is. Although this seems circular (the interest of an individual is dependent on the formulas it is involved in, and vice versa), it isn't. The interest of wffs is based on the current interest of the arguments, and does not change once it is set. Only the individual's interest changes.

A predicate's interest value may be set directly, or may be taken from whatever topics it indicates (see section on topics in previous chapter). When the system starts up, a number of interest values on predicates are already set up. The threshold for testing is set very low, so for most applications you won't need to change it.

Once adjudication of evidence is handled properly, this will give another bit of information which may be used as a termination criterion.

3.5.6.1 Interestingness

The "interestingness" values help the system decide what facts to follow up on (i.e. continue input-driven inferencing on), and which ones not to. Currently this is based on the interest level of the predicate involved in a wff, and the highest interest of any of its arguments. An arguments interest is the sum of all contributions of interest made by the wffs involving the individual. When calculating the interest of a formula, the current interest level of the individual is used, and once calculated, a wff's interest never changes, although that of the individuals involved does. Originally we also attempted to incorporate a salience factor into this calculation, "degrading" the interest level of an individual from the time it was last mentioned to the current usage, but this did not produce satisfactory results, and seemsed not important to the decision to continue inference. Salience will be important in determing referents of anaphora, but for inference termination we have rules it out for now.

For example, the interest level of

(lrrh pretty)

would be the product of *lrrh* 's current interest (say 10), and the interest level of *pretty* (from the topic *tp.appearance* - 3), giving 30. The contribution a wff makes to the individual is forced to be within 10 (otherwise the numbers get astronomical very quickly), so *lrrh* would be updated to have an interest of 20. Ideally, each argument in the literal will contribute some measure of interest, weighted perhaps. This has not been done yet because of other unresolved questions about interestingness.

Interest can be added to topics, and to types and other predicates using the **add-interest** command.

An inherited interest component is added to each wff from the parent that generated it - this is also used when testing for termination of forward inferencing. This inherited interest allows the system to "ride over" dips in the interestingness of a chain of inferences.

In addition, interest can be propagated backwards through a chain of causal connections - a formula is at least as interesting as anything it causes. So far this has not been investigated enough, and can be quite slow, so is turned off when the system starts up, but may be turned on by tweaking the parameter ***back-up-interest*** to **t** .

(add-interest *node interest*) [*function*]

Purpose: Adds interest levels to type and topic hierarchies.

Examples:

(add-interest 'human 5)

(add-interest 'tp.existence 10)

Remarks: Interest is propagated down the hierarchy until it reaches a higher interest level, or the bottom of the hierarchy. Note that this is most likely going to change in the next version.

3.5.6.1.1 Controlling Interestingness

Tweakable parameters which can affect interestingness calculations are as follows:

maximum-interest
Indicates the maximum interest value any object may have. This is initially set to 1000.

minimum-interest
Indicates the minimum interest value any object may have. This is initially set to 1.

deep-thought
a multiplier used on the interest during calculations to see whether to continue a particular inference line. Low numbers (< 1) indicate shallow thinking (i.e. prefer short inference paths), while high numbers indicate that longer inference paths are desirable. This is initially set to 1 (i.e. the interest alone makes the decision with no preference for inference path length either way).

back-up-interest
tells whether or not to back interestingness up causal paths. An interesting conclusion would then affect its causal precondition by increasing its interest. This is initially set off (nil), but may be set on for testing. There is still work to be done on determining exactly what constitutes a causal chain.

initial-charge
this is the amount to start the inherited interest component of new story sentences with. It may be necessary to give a new formula an extra boost to get it over the initial hump. However, our testing so far has not had this problem, so it is initially set at 0.

inherit-amount
this is the amount to decrease the inherited interest by at each step in an inference path. It is initially set to 5.

check-inherit
this flag indicates that forward inference should be allowed to continue based only on inherited interest values if the interest level of the particular formula is too low by itself. This is initially set to t (allow continuance).

wff-component
a multiplier used to indicate how much of a formula's interest should be used to update the interest value of its arguments. This is initially set to 0.1. Making the number higher makes the interest levels of new formulas involving existing entities increase more rapidly.

arg-component
a weighting used to indicate how much the interest of an argument affects the interest value of the entire formula. This is initially set to 1. Specific predicates may have weighting factors of their own for specific argument positions - this is an overal weighting to use after that.

operator-component
a weighting used to indicate how much the interest of an operator, predicate, or function affects the interest value of the entire formula. This is initially set to 1.

3.5.7 Controlling Input Driven Inference

The parameters which can be changed and affect input-driven inference are:

forward-full
Indicates whether or not to allow partial rule instantiation. If this flag is set to t , all variables must be matched in a rule before it is instantiated. The default is to allow partial rule instantiation.

forward-effort
Indicates how hard to try to verify rule antecedents during input-driven inferencing. Initially set to 0, if it is tweaked to something greater than 0, a simple backchaining effort will be attempted to verify the antecedent instead of just lookup or specialist evaluation.

interest-threshold
The minimum interest level (product of interest and probability) an inferred wff can have that will still enable more input-driven inferencing. This is initially set to 6, but may be changed using the *tweak* command.

key-threshold
The threshold of interest beyond which all keys will be included in the list of trigger keys for a rule.

salience
The maximum difference allowed between the current story array index, and the last index an entity in a wff was involved in. This ensures that we don't go making inferences with things that were talked about long ago. This is initially set to 10, but may be changed using the *tweak* command.

rule-forward
Indicates whether or not input-driven inferencing should be attempted on conditionals. Initially set on, but may be turned off using *tweak* .

story-forward
Indicates whether or not input-driven inferencing should be attempted on story facts. Initially set on, but may be turned off using *tweak* .

unify-sorts
Indicates whether sorts should be considered when unifying terms. Initially set on, but may be turned off using *tweak* .

Trace values which can be used to watch input-driven inference:

forward - traces forward inferences as they are made

forward-details - shows parent formulas when wff printed

infer-details - shows details of inference decision (for debugging)

forward-eval - traces evaluation of literals during forward inferencing

forward-test - shows comparisons between keys during forward inference

forward-access - shows accesses of formulas during forward inference

forward-class - shows classifications checked during forward inference

forward-qa - shows mini-question attempts during forward inference

forward-all - shows everything to do with forward inferencing, including *forward, rules, access, forward-rejction, forward-test, forward-unify, entry-time* and *forward-eval* .

forward-int - shows interesting parts of forward inference (everything except classification)

forward-min - shows the inferences made, and evaluations during forward inferencing

entry-eval - traces verifications done during input

entry-time - keeps track of how long it takes to enter a story formula, including all the forward inferences.

entry - shows interesting parts of story entry, including *forward* and *entry-time* .

entry-eval - shows evaluations during input of formulas

entry-min - shows inferences made through forward and mp inferences.

entry-int - shows interesting stuff for both forward and mp inferences.

entry-int - shows everyhting possible about entry of formulas including everything about both forward and meaning postulate inferences.

3.6 Input-driven Inference vs Goal-driven Inference

Given the problem of determining when an input-driven inference should be kept, one might ask why we shouldn't just turn off forward inference, and get all the inferences using goal-directed inference (which might take longer). It seems at first sight reasonable to be able to do so. But as a matter of fact, there is a rather good argument against it, as follows. (In outline, the argument goes like this: when we work out in detail what would be needed to get complete goal-driven inference with out input-chaining, we find that we would need to index "facts" in a way that can only be done reliably by input-chaining on those facts!)

We begin by considering how we would prove a goal like *(LRRH pretty)* , given [LRRH (very pretty)) . Even if we had indexed the latter fact under (LRRH pretty subject), it's not at all clear how this would allow us to infer the goal. One suggestion is that we trigger forward MP-based inferences on retrieved facts, but that is a rather complex strategy and also one in violation of the assumption that forward inference has been turned off. So, we would need to chain backward on MPs such as the one that says that necessarily, if something is (very P) then it is P.

But, though such MP-based goal chaining may well be useful at times, it should not be done indiscriminately. We don't want to chain back from *(LRRH pretty)* to *(LRRH (very pretty))* , *(LRRH (very (very pretty)))* , *(LRRH (very (very (very pretty))))* , *(LRRH (extremely pretty))* , *(LRRH (exceptionally pretty))* , *(LRRH (unusually pretty))* , *(LRRH (amazingly pretty))* , etc. If we have retrieved *(LRRH (very pretty))* , we would like to use just those MP's that promise to make a connection between the seemingly relevant retrieved fact and the goal; i.e., we would like to use just those MPs which have a positively occurring part matching a goal key and a negatively occurring part matching (part of) a seemingly relevant retrieved wff. This could be implemented by only using those MPs for goal-chaining which are simultaneously indexed (positively) by the goal and (negatively) by a fact seemingly relevant to the goal. Moreover, we would use a "simultaneous instantiation" based on the goal and the relevant

fact. this could all be done with minimum fuss by forming new goal *(relevant-fact -> goal)* and then doing ordinary goal-chaining (but allowing MPs for goal-chaining).

But all this presupposes that we have indexed a formula like *(LRRH (very pretty))* under (LRRH pretty subject), and that raises the question of how we do such indexing for operator- embedded predicates. One approach would be to make a careful study of predicate operators, and come up with a comprehensive scheme for classifying all complex predicative expressions. For example, we might distinguish operators like {very, extremely,...} (i.e., intensifiers) from ones like {somewhat, moderately, sort-of,...} (i.e., hedges), and from ones like {make, convert-to, ...} (i.e., verbs of synthesis), and from ones like {become, turn-into,...} (i.e., verbs of becoming), etc. However, that's a hazardous approach, likely to lead to unexpected "holes" in inference behavior, because it is so dependent on anticipating what classes of operators will "come along" as the system is further developed, and on the user understanding the classification schema and correctly classify- ing all operators in accordance with it.

A much more "robust" approach, not dependent on explicit operator classification by the user, would be to base the classification on the MPs involving the operator. For example, the reason it makes sense to classify *(LRRH (very pretty))* under (LRRH pretty subject) is precisely that there is a MP which says that whatever is (very P) is P. So we should use that MP to automatically obtain the classific- ation of *(LRRH (very pretty))* . So we should match this formula to negative keys in MP's, just as if we were going to do a forward inference, to derive its classification. Now at first glance it appears that we could do this without actually performing any inferences, i.e., we just do some sort of matching on the MP. But consider input facts like *(LRRH (very (very pretty))* or *(LRRH (very (Lx ((x pretty) and (x friendly)))))* or *(LRRH (became (very pretty)))* , etc. (ignoring episodes, obviously). From such examples it becomes clear that the "matching" on the MP would implicitly have to duplicate multiple input-chaining steps; e.g., it recognizes that *(LRRH (very (very pretty)))* implies *(LRRH (very pretty))* , which in turn implies *(LRRH pretty)* and that's why it should be classified under (LRRH pretty subject); similarly it recognizes (from one MP) that *(LRRH (became (very pretty)))* implies *(LRRH (very pretty))* , and (from another MP) that this in turn implies *(LRRH pretty)* , and hence the original formula should be classified under (LRRH pretty subject); etc.

But it would be silly (and hard) to duplicate the input-chaining mechanism in some implicit way in the classifier! We might as well do the input chaining, and having done it, we might as well retain the conclusions, and classify these directly and straightforwardly based on their explicit predicates (complex or atomic). Q.E.D!

3.7 Problems

Inference merging has not been implemented yet, but is particularly important for connecting up story sentences. Thus **EPILOG** is not capable of finding the causal connections between story sentences yet.

3.8 TroubleShooting

This section describes how to track down some of the more common problems associated with input driven inference. If, after trying the following, you still cannot solve the problem, feel free to send it to one of the authors to solve.

3.8.1 What to do if a Desirable Inference is NOT made

When an inference that is expected is not made, the user should run the system until it gets to the formula that he/she was expecting to trigger the inference, and trace the system operations to see what is happening. To get the system to try again to make the inference, use the command **reassert** with the formula name of the one you expected it to make the inference from.

First figure out what rule you expected to fire on the input (inference or initial assertion). Then trace *forward-int* and *forward-test* (*mp-int* and *mp-test* if it is an mp inference you were expecting). Get the system to try making inferences with the input formula again using **(reassert** *wff-name* **)** . Watch and see if the system ever even *tries* to compare the input formula with the expected rule.

There are a number of reasons that could possibly explain why an inference isn't made:

The applicable rule was not found
If no comparison with the expected rule occurs, look to see where the rule is classified (use *(classify rule-name)* or *(mp-classify rule-name)* for an mp). Try again, tracing *forward-class* and *forward-access* (*mp-class* and *mp-access* if this is an mp inference) to see if it ever looks under the classifications for the rule. If it does, check the next possible problems; otherwise continue here.

One possible problem could be that there is a type mismatch between the input and the rule so the formula is never found. This would mean that the first part of the classifications checked never matches the first part of the classifications the rule is stored under. This may also be an ordering problem - remember that an entity's type must be input before any other information about that entity. If ordering is the problem (especially types), ensure that input types are done first, and that rules which create existentially quantified entities in their consequents have types in the restriction clause of the existentially quantified expression.

It is also possible that the system has missed a "trigger" key in the rule or input, and the predicate involved should be set up differently so that the system thinks it is more important. This would mean the second part of the classifications doesn't match any of the ones the rule is stored under. The rule itself may not have an important key selected as a trigger key. If you look at the classifications for the rule, and it seems that there is nothing for an important part of the rule, this may be the problem. This means that you'll have to restart from before that rule was even entered, however. You can force the system to make all keys trigger keys by setting ***key-threshold*** to 0. If the desired inference is made, you'll know that trigger key selection is the problem, and can boost the interestingness of the predicate involved (using command **add-interest**) so that it is selected as a trigger key, and then run normally (with the key threshold at its normal level).

The applicable rule was found but had been applied previously
To prevent computing transitive closures, EPILOG will not allow any rule to apply more than once in an input-driven inference chain. If the rule was already used in this particular chain, it will not be attempted again. If it is essential that this inference be made, perhaps another, similar rule can be entered which is more specific to this particular stage in the chain. Meaning postulates, however, CAN be applied more than once, so this should only occur with regular knowledge rules.

The applicable rule was found but did not match the input formula
Trace *forward-test* - and when you see the comparison between the rule and the input, watch to see which keys of the rule it compares with. If none of the comparisons succeed, the rule will not be used. If the key

that should have been compared with in the rule is not even attempted, it is a trigger key problem, and can be solved as discussed above. If the key is being compared, but the comparison is not successful, there are several possibilities depending on the actual key. A common problem is that the rule is expecting ** and the input has only * - if this rule really has to succeed the antecedent must be made less strict. Another problem could be that appropriate sorts have not been assigned to the input entities - but this rarely causes problems here.

The appropriate match was made, but the rule didn't completely fire
In this case, other information in the rule was expected to be evaluated, and it was not available. If only full instantiation is allowed (i.e. *forward-full* is t), no inference will be made, otherwise a strange looking one like *((wolf1 grey) implies (wolf1 fierce))* will be. If you are tracing *qa-int* , you should be able to see it trying to find formulas to match the other trigger keys, as well as trying to verify the non-trigger keys. A failure for any of these could be at fault, although a failure to match or evaluate something is certainly not always a problem (the desired inference won't be matched or evaluated!) If the system is looking up something to match another trigger key, and the information isn't currently available, the rule will be fired again when it does become available. If this won't happen because it was expected that the information should already be there through another inference, the problem is with THAT (unmade) inference, not this one. If the information IS already available, you should be able to see (via the tracing) if it finds it and the comparison fails (a comparison problem like above) or doesn't find it (classification). Non-trigger keys are evaluated either by simple lookup or by a specialist. If a particular key was expected to be evaluated by a specialist, trace that particular specialist's actions to see if the information is there and why it does not answer. If the appropriate sorts were not on the arguments when the information was first entered, the specialist may not have accepted the input, which would explain why it is not used when you expected it (especially notable with the time specialist - all temporal input must have sort *time* or *episode* on the arguments to be recognized by the specialist).

In the case of meaning postulates, all meta-level variables must be matched before the rule will fire, so this may explain why an expected inference is not made.

The system just stopped looking for rules before getting there
Make sure that it didn't just run out of classifications - when tracing *qa-access* , you would see it enter the formula, and then NOT access anything to apply. If it does access classifications looking for rules, the problem is that it doesn't find the rule, not that it just stops.

If it just stops, this suggests an interestingness problem - the system has decided that the information leading to the desired inference isn't interesting enough to continue with. To test this, set ***interest-threshold*** to something very small (e.g. -10), and try again. If you get the inference, then you know this is the problem. That can be fixed by changing the interest level of one of the predicates along the chain of inferences before it stopped. If this doesn't get the inference for you, then the problem is one of the above - either an appropriate rule is not being found, or is not being compared successfully. This problem is quite unlikely to occur, unless the interest threshold has been boosted beyond its default.

3.8.2 What to do if an Undesirable Inference IS made

Sometimes the opposite problem occurs - an unexpected, and undesirable inference is made. There are several types of these:

Those which indicate some information was missing
These are like the *((wolf1 grey) implies (wolf1 fierce))* above or *((wolf1 wolf) or (wolf1 fox))* , where expected information was not present. Usually this is caused by types being entered after other information about entities, or expected inferences not being made. See the section above on when a rule doesn't completely fire for more tips.

Conditional Inferences
The system generates partial rule instantiations, which often are rules themselves. These may then be applied like regular input rules. Although they seem annoying and useless to the user, often it is these that allow the later, more interesting inferences to be made. An alternative is to use total rule instantiation (***forward-full*** must be *t* and ***key-threshold*** must be 0). However, this doesn't always give the best results either. If the system really is generating "garbage", it may fit into the next category.

Garbage and incorrect inferences
These usually can be traced back to strange rules or rule combinations. Look at the inference history of the strange inference (using **(display 'infer** *wff-name* **-f)**), and see if the rules that applied were applied correctly (i.e. the antecedents were true, etc). If so, look at the rules themselves to see if they make sense. Incorrect bracketing can cause a rule to say something other than what was intended so check carefully. If the rule seems to have applied when it shouldn't have, look at the other formulas involved which triggered the rule - maybe one of them came from an inference from another rule which didn't make sense. Another possibility is that the hierarchies are tangled incorrectly, making some entities of a type which you didn't anticipate.

Chapter 4

Questions and Queries

4.1 Using Equality Information

When equalities have been entered into the system (usually with *equal*), they are available for use by the main system to use in looking up formulas, and comparison and unification of formulas. The set specialist detects and stores the equivalences in a hash table accessible to EPILOG. When retrieving information for a given classification, it also retrieves information stored under equivalent classifications, by using the equivalence class for the first element of the classification. When comparing two formulas to see if one supports the other, the arguments are compared taking equivalence classes into consideration. The same is used during unification when two constants are being compared.

The equivalence information is modal-predicate sensitive. The equality-specialist, which detects and stores the equalities, and handles the modal embedding properly. To ensure that all modally embedded information is available for all equivalent items, formulas which are modally embedded are duplicated in specialists - they are stored under all equivalent subnets. This is not necessary in the main EPILOG system. Evaluations and comparisons using specialists need only check one subnet though - as it is guaranteed that any of them will have the relevant information.

4.2 Queries

The following functions are used only to display information in the knowledge base, and require no inference. Questions which do required inference are handled by the question answering routines in the next section. The "display" commands are intended to display information in a pretty form for the user; the "retrieval" commands return lists of information which can be used either by the user or another Lisp routine.

4.2.1 Display

The following display options may be used with the **display** command:

(display 'infer *wff-name+* **)**

Displays the inference path which led to the given wff names. If *-brief* or *-b* is specified, only the last inference is shown. If *-full* or *-f* is specified, the inference path right back to the input formulas

is shown. If you are tracing *'forward-details* this information is automatically displayed for each inference. This display option allows you to see the inference process "after the fact". (Note, this was previously achieved using the functions **display-infer** and **display-infer-short** . These functions are still available but this option provides a more uniform and easier to use interface). Some examples:

(display 'infer 'wff1 '-f)
will display all inferences leading up to the formula *wff1* .

(display 'infer 'wff2)
will display the particular inference which generated the formula *wff2* .

(display 'formula *wff-name+* **)** or **(display 'wff** *wff-name+* **)**
Displays the formula (list form) corresponding to the given wff names. For example:

(display 'formula 'wff1)
might print out *WFF1: (lrrh girl)*

(display 'prob *wff-name+* **)**
Displays the lower subjective probability associated with the wff names given. For example:

(display 'prob 'wff1)
might print out *Probability of WFF1 is 1*

(display 'node *entity topic* **)** or **(display** *entity topic* **)**
Retrieves and prints information classified under a concept from the topical classification table. If *topic* is not specified, all topics are printed, otherwise only wffs under that topic, and topics beneath it are printed. If *-full* or *-f* is specified, subnets for the entity are also printed. If *concept* is a skolemized variable, it will be replaced by its skolem constant for the retrieval. This is the default for the **display** command in EPILOG, so no option is necessary (i.e. you can eliminate the *'node*). Note: this function was originally performed by the routine

(display-concept *concept topic super sub print-subnets subnet* **)**
That routine is still available, but the display option is easier to use. Examples:

(display 'lrrh)

displays all information about *lrrh* , except her belief subnets.

(display 'lrrh '-f)

displays all information about *lrrh* , including any belief subnets associated with her.

(display-concept 'animal 'tp.spec)

displays all formulas which assert that something is an *animal* .

(display-concept 'animal 'tp.spec '-f)

displays all formulas which assert that something is an *animal* , or any subtype of animal.

(display 'wffs)
retrieves and prints, with classifications, every formula that has been entered. The main classification is used to organize and display the output. If *-full* is specified, subnets will also be printed.

(display 'wffs-by-topic)
retrieves and prints, with classifications, every formula that has been entered into the topically organized table. This is all the formulas except meaning postulates. If *-full* is specified, subnets

will also be printed. The topical classification is used to organize and display the output. Note: the previous two display options (*wffs* and *wffs-by-topic*) were previously handled by the routine **show-everything** . This is still available, but the display interface is much easier to use.

(display 'input-array) or **(display 'input)**
Displays in input order story sentences that have been loaded so far as stored in the knowledge base. Previously this was handled by the command **show-input-array** , which is still available.

4.2.2 Retrieval

(retrieve *concept topic super sub print-subnets*) [*function*]

Purpose: Retrieves wff names of propositions classified under a particular concept from the topical classification table.

Syntax: Retrieves the propositions for *topic* and *concept* . If *print-subnets* is t, the contents of subnets will be exhaustively retrieved. If *sub* is t, topics beneath *topic* in the hierarchy are also printed (also concepts beneath *concept* in the type hierarchy); if *super* is t, concepts above *concept* are also included. If no topic is specified, all topics are retrieved. *Retrieve* may be abbreviated to *ret* .

Examples:

(retrieve 'lrrh)

returns a list of wff-names which are about *lrrh* , not including beliefs.

(retrieve 'animal 'tp.spec nil t)

returns a list of wff-names which assert instances of *animal* , and any subtype of *animal* .

(get-everything *subnet &optional (print-subnets t)*) [*function*]

Purpose: Retrieves a list of all wff names in a subnet

Syntax: *subnet* may be **(environment-main-net *environment*), (environment-topic-net *environment*)** , or any other subnet. Note that if you use a topic net, meaning postulates won't be included. Get-everything returns a list of all the propositions in *subnet* , including its subnets if *print-subnets* is t.

Examples:

(get-everything (environment-topic-net *environment*) t)

returns a list of every formula name, including modal formulas, but no meaning postulates.

4.3 Asking Questions

These routines are to be used when asking a yes/no or wh question. Numerous system parameters are available to change for maximum control of the system. These are described in a later section.

(question *&optional formula number effort*) [*function*]

Purpose: To answer yes/no and some wh questions.

Examples:

(question '(E x (x girl) (x happy)))

(question '(WH x (x girl) (x happy)) 2)

(question '(WH x (x girl) (x happy)) 2 1)

(q)

Remarks: *Question* may be abbreviated to *q* . This uses a goal-directed inference mechanism which will be described in detail shortly. If no *formula* is given, the system continues working on the previous question. For wh-questions, if *number* is specified, the system stops after finding *number* answers. If *effort* is specified (must be an integer between 0 and 3), it controls how hard the question-answerer will work; otherwise ***question-effort*** is used. An effort of 0 specifies lookup and specialist evaluation only; 1 indicates that splitting of subgoals may be done; 2 allows inference as well; and 3 allows assumptions to be made. The answer returned is a list of solutions. For yes/no questions, each list consists of Yes or No, the wffs involved, and the probability. For wh-questions, each list consists of the set of entities that matched the variables quantified by WH in the question, as well as the wffs involved, and the probability.

If you want the system to concentrate on a specific answer which you expect, use command **proof-q** (or **pq**) to ask a question whose answer you believe is YES, and **disproof-q** (**dq**) to ask a question whose answer you believe is NO. These should only be used for debugging purposes. Parameters are the same as for **question** .

Some display options which may be useful:

(display 'agenda *number* **)**

prints access and subgoal actions from the question answering agenda in a readable format. This should be used when the question answerer has stopped in the middle of answering a question and it is desirable to see what options it is considering. If *number* is not specified, all items on the agenda will be printed; otherwise only the top *number* of them. Note - this was previously accomplished using the **print-agenda** command. For example:

(display 'agenda)

will print out the entire agenda.

(display 'agenda 5)

will print out only the top 5 entries in the agenda.

(display 'subgoal *subgoal* **)**

prints a trace of the proof process for a given subgoal in a readable format. It shows at each stage the subgoal in question, and the further breakdown into subgoals, and how they should be combined. This may be called in the middle of a question or when the question has finished. *Subgoal* is optional for a successful questioning attempt, or may be specified. It must be one of *'proof* or **proof** (the proof attempt) or *'disproof* or **disproof** (the disproof attempt). If the question terminated successfully, *'disproof* will be the default if the answer was NO, otherwise *'proof* . If -full or -f is specified, some extra debugging information will also be printed. If debug is not specified, and the question attempt finished successfully, only those subgoals in the subgoal tree on the "proof path" (the ones that contributed to the answer) will be printed. Otherwise all subgoals

in the subgoal tree are printed. Note - this was previously accomplished using the **print-subgoal** command. For example:

(display 'subgoal 'proof)
displays the subgoal corresponding to the proof attempt.

(display 'subgoal)
displays the proof subgoal if the answer to the current question was YES, and the disproof subgoal if the answer was NO. Note that no subgoal will be printed if the question hasn't been answered.

(display 'subgoal 'disproof '-f)
displays extra information for the current disproof attempt.

(display 'proof)
is just a shorter form for *(display 'subgoal 'proof)* .

(display 'disproof)
is just a shorter form for *(display 'subgoal 'disproof)* .

(display 'question) or **(display 'q)**
displays the current question.

(display 'solutions) , **(display 'solution)** , **(display 'answers)** , or **(display 'answer)**
repeats the answers for the last question.

Useful global variables and constants:

solutions
This contains the list of all answers found during a question answering attempt, the wffs used to get the answer, and their probabilities.

question
The current question being asked.

The system can respond to questions which can be answered Yes or No. The same syntax is used to ask questions as is used for asserting formulas. For example,

(q '(lrrh pretty)
would be answered Yes (assuming the input given earlier),

(q '(E x (x girl) (x pretty)))
would be answered Yes, and

(q '(A x (x girl) (not x pretty)))
would be answered No.

The system first tries the fastest, easiest method available - a simple lookup, or specialist evaluation. If this fails, the current effort level governs how hard the system will try to answer the question. Besides the effort level, there are numerous controls set up so that the user can control the question answerer if he so desires (see the section on controlling the question answerer).

The system takes the question and its negation as the *proof* and *disproof* attempts, respectively (note that the old system used the negation of the question as the *proof* attempt, and the question itself as the *disproof* - but the new version does not work by contradiction because it is harder to follow the proof) .

It then tries to prove both simultaneously - if the proof subgoal can be proven, the answer to the question is YES; if the disproof subgoal is, the answer is NO.

Each subgoal (top level or result of an inference) is put through a natural-deduction-like splitting process, where it is split into simpler subgoals. Quantified conditionals and logical conditionals are handled by assuming the antecedent, and adding a new subgoal to prove the consequent. Logical (independent) *and* s are split into a separate subgoal for each component of the conjunction, and these are all added as new subgoals. For disjunctions, the negation of all except the first part is assumed, and a subgoal is added for the first member of the disjunction. Nothing else is considered splittable.

When a subgoal is no longer splittable, access actions are created for it and added to an agenda of things to do when answering the question. The system then looks up rules and facts in the knowledge base and compares them against the subgoal, creating new subgoals, until eventually one of the subgoals is proven true (or it runs out of time). MP's may also be used (in reverse) to help answer the question.

When finished, the answer is printed, as well as an indication of how hard the system had to work to answer the question.

Sometimes the system can find more than one inference path leading to an answer, and these may support or contradict each other. EPILOG adjudicates the answers to determine what the answer is and what the probability should be. However, currently it cannot determine inference path subsumption or specificity of the inference paths, so it does not take that into account when doing the calculations. This can lead to probabilities higher than they should be.

Responses to questions may optionally be "said" in English - see the section called *Response Generation* for details.

The current method has changed significantly from previous versions, which tried lookup and specialist evaluation, then simple backchaining, and finally proof by contradiction to answer the question. The new method also tries the lookup and specialist verification, and has both a *proof* attempt and a *disproof* attempt, and uses an agenda, but the backchaining and proof by contradiction have been replaced by a new method which uses a natural-deduction-like breakdown of subgoals, along with a modified backchaining approach.

An effort level controls how hard the question answerer works. This may be specified for an individual question (when the **question** routine is called), or globally using the tweakable paramter ***question-effort*** . The effort levels are as follows:

0: Lookup/verify only - the system does a simple lookup of the question, or may use a specialist to evaluate it.

1: Allow natural-deduction-like breakdown of subgoals. This also allows the use of the agenda. For questions which consist of several parts, or with variables to be matched, this is the minimum level required (especially for wh-questions). No inference is actually done, just retrieval of facts and unification.

2: Allow inference. Not only facts are allowed to be used, but also conditional statements (rules and to a limited extent, meaning postulates).

3: Allow assumptions to be made. For some kinds of subgoals, the natural way to solve them is to assume one part, and try to prove the rest (for conditionals and disjunctions). These assumptions are actually entered (temporarily) into the knowledge base, and input-driven inferencing is optionally attempted on them. This is the most expensive operation because the assumptions must be retracted

when the question is finished. Unlike ECoNet, ONLY these assumptions are actually entered into the knowledge base - the subgoals themselves are not.

If a simple lookup or specialist evaluation fail to answer the question, the system tries to simultaneously prove the question (the *proof* attempt) and its negation (the *disproof* attempt). If the question itself can be proven, the answer is obviously YES; if its negation, the answer is NO. Note that the previous version worked by contradiction, and the question was the *disproof* attempt, and its negation, the *proof* attempt. The current version could also have used contradiction, but it is more difficult to follow the proof, and also requires more retraction later.

An agenda of actions is used to keep track of what is left to try. There are two types of actions on the agenda: *access* actions, and *subgoal* actions. *Access* actions look for more wffs to compare against a set-of-support *proof* or *disproof* wff. Both regular backward inference and contrapositive inference are supported here (i.e. matching the consequent, resulting in the antecedent becoming the new subgoal; or contradicting the antecedent, resulting in the negation of the consequent becoming the new subgoal). If any wff is useful for either direction with that set-of-support wff, a subgoal action is added to the agenda for it. Some checking is done to try to avoid runaway recursion, especially with transitive rules. Also, meaning postulates (and simplification schemas) are retrieved and applied (in the backward direction only). These are highly penalized in the agenda ordering so that they occur at the bottom of the agenda, as a last resort.

Subgoal actions prepare and add a new subgoal using the two wffs and the comparison information from the access (substitutions, residues, etc). After subgoal splitting, access actions for this new subgoal are added to the agenda.

When a subgoal is first entered, it may be split into smaller subgoals. The answers to the smaller subgoals are then combined if necessary to get the answer to the parent subgoal. Subgoal splitting continues, building a subgoal tree, until a "leaf" subgoal is reached (unsplittable), and access actions are added for that subgoal. A "difficulty" measure is added to each subgoal depending on the split done.

The agenda, at any time, contains actions for both the *proof* and *disproof* attempts, ordered by depth of search so far, interest level of the particular subgoal, complexity of wffs involved, the "difficulty" value determined by subgoal splitting, and probability. The system takes one action from the agenda at a time, and does it. If subgoal action result is YES or NO, the answer is backed up to the parent subgoal. This may or may not solve the parent subgoal as well - if so, the answer is backed up again. If the top level *proof* subgoal answer is YES, the answer to the question is YES; if the top level *disproof* subgoal answer is YES, the answer to the question is NO. We stop going through the agenda if any one of the following stopping conditions are met:

1) There are no more items on the agenda.
   There is not enough information to answer the question.

2) The maximum number of iterations has been reached.
   The tweakable parameter ***qa-iterations*** is used to control the search process so that if a question cannot be answered, we stop fairly soon instead of going on forever. This can also be used to single step through the system and inspect things as they happen.

3) An answer has been reached, and it is of sufficient probability.
   An answer has been reached and its probability is greater than the threshold ***question-threshold*** . This threshold allows us to ignore answers that are not very likely, and to keep searching for others. (Note, if the minimum effort ***minimum-effort*** has not yet been expended, the system will continue looking for answers until it has been).

4) ***max-wh-difference*** has been specified, and any further searching will go to depths lower than this flag indicates (wh-questions only).

If the system stops because the maximum number of iterations has been reached, the process can be continued by invoking *question* with no formula. If it stops because an answer was reached, we can either accept that answer, or tell the system to go on and find others (again, by invoking *question* with no arguments). The easiest answer to obtain is the one that comes back first; this may or may not be the one with the highest probability.

If an answer to a YES/NO question is found, and its probability is less than 1, there is a possibility that there is more evidence out there to support or contradict the answer. One should be able to examine the agenda to see if the information there is likely to lead to another answer. Since that is quite difficult to do (you might as well continue trying to answer the question!), a ***minimum-effort*** parameter has been introduced. This is the minimum number of iterations to use when answering a question. If an answer is found (of probability less than 1) and this many iterations (the default is 5) have not been used, the system will continue trying to find another answer until that minimum is reached. Multiple inference paths for YES/NO questions are combined to give the correct answer and probability.

For wh-questions, often all the answers are obtainable with the same amount of work each - i.e. the same depth. To prevent the system from "trying too hard" to get more answers, the ***max-wh-difference*** flag was introduced. This flag is the maximum difference between the depth required to get the first answer, andthe depth required to get any subsequent answer. If set to nil, this flag has no effect at all on the search. Otherwise, once the first answer is detected, the agenda is reordered in more of a "breadth-first" manner, and search continues until a depth lower than the maximum is detected on the agenda. This is most efficient if the flag is set to 0 - which appears to give the desirable results in most cases (so it is the default). If the system stops because of this, doing (q) again will resume the search (at least for a while).

Actions on the agenda are ranked based on their depth (rank) (high rank preferred over low), type (subgoal preferred over access) (regular preferred to contrapositive), a combination of interest and complexity (the interest is divided by the complexity to "normalize" it, and a high value of this ration is preferred to a low value), difficulty (low preferred over high) and probability (high preferred over low). How much each of these contributes to the ranking is handled by tweakable parameters ***qa-access-weight*** , ***contra-weight*** , ***rank-importance*** , ***interest-importance*** , ***difficulty-importance*** and ***prob-importance*** . These are initially set up so that rank, depth and type are highly important, followed by difficulty, interest and complexity. These may be fiddled with by the user to the optimal performance for a specific group of questions if desired. Setting one of these to 0 eliminates that item from consideration in the ranking. Higher ranks, i.e. deeper subgoals are intially preferred. But if the system goes off on a tangent, it can go deeper and deeper getting nowhere. To prevent this, the system initially prefers deeper depths, until they get to half of a "maximum" depth ***qa-depth*** . After this they become less and less important, and when the depth reaches the maximum it will not be considered at all in the calculation (which should make the subgoal quite undesirable, unless it is one of the few remaining). Since the initial probability is not known, **initial-prob** is used as the probability to order initial access actions with. This is initially .75, and should be less than 1 so that the initial subgoals are not favored over later ones simply because the later ones have used knowledge base formulas with probabilities less than 1. This does not affect the probability of answers - just agenda positioning.

Interest calculations may also include "inherited" interest - similar to what the input-driven inference mechanism uses. Here the maximum of the agenda item's interest and its inherited interest is used in the interest-complexity calculation.

For access actions, the ranking also includes the complexity of the testing literal selected from the proposition, with high complexity being preferred over low, and an additional weight for comparisons which involve residues.

Even with carefully chosen parameter values, it is still possible for the system to "go off on a tangent" and work completely on on proof/disproof attempt to the complete exclusion of the other. If half of ***qa-iterations*** have been used up (with no answer) on just one of the attempts, the top agenda item for the other attempt is extracted from the agenda, "primed" to make it more interesting, and the system attempts to work with that for a little while. This doesn't happend very often, but it can make the difference between answering a question or not answering it. This is only done for yes/no questions where ***qa-iterations*** is greater than 5.

4.3.1 Subgoal Splitting

Subgoals are put through the splitting process whenever added, whether the subgoal is a top level subgoal (the question or its negation), the result of a previous split, or a new subgoal resulting from an inference. The subgoal tree is built by recursively splitting subgoals using the following:

1) For a conditional subgoal, the antecedent is assumed, and a subgoal is created to prove the consequent. If the effort level is set so that assumptions are not allowed, conditional subgoals are not even attempted.

2) For a conjunction where all the parts are "independent" (i.e. none of them contain variables quantified in another part), each part becomes a new subgoal, and the "child" subgoals' answers will be combined with *and* to get the parent subgoal's answer.

3) For a disjunction (again with "independent" parts), the negation of all except the first part are assumed, and a new subgoal is added to prove the first part. If the effort level is such that assumptions are not allowed, new subgoals will be added for each part, and their answers will be combined with *or* to get the answer to the parent subgoal.

4) All other subgoals are considered non-splittable "leaf" subgoals. Access actions are added for these to the agenda.

Assumptions are actually entered into the knowledge base during this process. They are flagged as assumptions, and may only be used in verification or inference by the subgoal resulting from the split where the assumption was made, and any descendants of that subgoal. Forward inference may optionally be attempted on the assumption (if ***goal-forward*** is t), and any inferences made will also be flagged as assumptions, and will be available to any subgoal the original assumption is available to. The assumptions and their inferences are all retracted when the question is finished.

A difficulty measure is added during the splitting process - splits which requiring assumptions have the highest difficulty; those requiring no splitting have the lowest difficulty. This difficulty measure is also cumulative - the difficulty of a particular subgoal is the sum of the difficulty measure obtained in each split required up to that subgoal. The difficulty levels are handled by tweakable parameters: ***quantified-difficulty*** (value 20) is the difficulty assigned to non-conditional quantified subgoals; ***conditional-difficulty*** (value 30) is assigned to conditional subgoals solved using the "assume antecedent - prove consequent" method; ***split-difficulty*** (value 10) is for subgoals which are split into several parts (simple and/or splits), and ***assume-difficulty*** (value 30) is for disjunctive subgoals which are solved by assuming the negation of one of the disjuncts.

Each node in the subgoal tree also contains information on how the answers returned from its children should be combined (*and* or *or*).

4.3.2 Access Actions

Each access action consists of a set-of-support wff, a testing key from that wff, and classifications to use to find formulas to compare with the literal. For each formula retrieved, the trigger keys are taken from it and compared with the testing key. Positive trigger keys are compared for compatibility, negative for incompatibility (regular and contrapositive backward chaining). If the obtained wff is a meaning postulate, only its positive keys will be checked.

We go through this list of classifications, retrieving clauses and comparing against the set-of-support key until:

1) A wff is found which compares favorably with the set-of-support key.
   Subgoal actions are added to the agenda with information from this comparison (there may be more than one - several keys from the retrieved wff match, or with different unifications from the specialists). The remaining wffs from that retrieval, as well as remaining classifications are put back onto the access action, and it is placed back on the agenda in a lower position.

2) The maximum number of wffs to test per access (***max-wffs***) has been reached.
   The remaining wffs and classifications are put back on the agenda just as for a matching wff (but no subgoal is added).

3) The maximum number of classifications to retrieve per access (***max-class***) has been reached.
   The remaining classifications are put back on the agenda as above.

Anytime a classification has been exhausted, if appropriate, we add "super" classes (using the parent or types (if a constant) of the first part of the classification and the same topic), and "sub" classes (using the children and instances of the first part of the classification, and the same topic). For example, a super access for *(wolf . tp.social)* would be *(warm-blooded-quadruped . tp.social)* , and sub accesses for *(creature . tp.appearance)* could be *(human . tp.appearance), (animal . tp.appearance), (c1 . tp.appearance)* , etc.

These access actions are similar to the ones used by **ECoNet** . If we do only one classification per access, and allow unlimited comparisons with wffs per access, we have essentially the same action, except that **EPILOG** stops after the first match and attempts to use it, whereas **ECoNet** had to compare against all clauses in a classification at once. This sometimes was rather annoying since it would compare against a whole pile of clauses, any one of which was good enough to answer the question. **EPILOG** avoids that, without losing any possibilities for comparison, since they are still waiting on the agenda. The maximum wffs and maximum classifications allowed is an attempt to equalize the actions on the agenda somewhat - an access that retrieves nothing is very fast, and gives no results. We may need numerous of these actions to get anywhere. However, if the same number of actions are allowed where lots of comparisons are being done, they take much longer. This means that setting the number of iterations as a method of controlling how long a question should be allowed to take will not work quite the way we want. By changing those maximums, we can get more control.

Access actions are now ranked based only on the set-of-support formulas they are looking for wffs for, unlike **ECoNet** , which also took into account the number of propositions available through an access. If the testing literal is negated, it is placed further down on the agenda than if not (using ***contra-weight***

for the amount since it is more likely that it will be incompatible with something, and inompatibility is only used for contrapositive inference). The ordering of access classifications is an independent problem from the ordering of agenda items. Access classifications are ordered in the following manner: classifications involving individuals firsts, followed by those involving a type predicate, followed by those involving sorts, followed by all others. If the first classification in a set to look at is a type classification, and the subgoal is existentially quantified, the system will look at the instances and subtypes of the classification before the classification itself, and before the other classifications.

If a rule has been applied once, it may be applied again, but only if the same trigger key is used. This prevents the same rule from firing in both the forward and backward directions, which could cause the system to oscillate forever. Access actions must also ensure that we are not recursively digging into a hole, which can happend especially with transitive rules. To prevent this from happening, the system insists that for any particular rule to be used more than once in a proof, a non-conditional must have been applied between the two uses. For example, suppose we have as facts *e1 before e2 before e3 before e4* , and rule

(A x (A y (x before y) (A z (y before z) (x before z))))

We can apply the rule to question *(e1 before e4)* , resulting in subgoal

(E x (e1 before x) (x before e4))

If we immediately apply the rule again, we'll get subgoal

(E x (e1 before x) (E y (x before y) (y before e4)))

which is going nowhere fast! However, by forcing it to apply a fact (e.g. *(e3 before e4)*) first, we can prevent this - the resulting subgoal would be *(e1 before e3)* , and we can apply the rule again, and then *(e1 before e2)* to answer the question.

4.3.3 Subgoal Actions

These actions correspond to the *resolution* actions in **ECoNet** , although the techniques used are somewhat different here.

Subgoals are created using "goal reduction" - the dual of rule instantiation. The matching literal in the retrieved formula is replaced by YES (for negatively occurring literals), or NO (for postively occurring literals). Substitutions and normalization are done, and the result negated. The resulting formula then replaces the testing key in the original set-of-support wff.

This new subgoal is then verified - if it evaluates to YES, we have an answer for the current subgoal. (If the answer is NO, we do have an answer, but it is not of any use to us - just because one set of circumstances led to a NO, doesn't mean there isn't a set that will lead to YES. If the answer is NO, the other attempt will find it.) This process may also simplify the subgoal. If we have an answer for this subgoal, the answer is propagated to the parent subgoal, which will examine it and the answers from its other subgoals so far, and use its combination information to see if the parent subgoal is solved. If so, the answer is propagated to its parent, and so on. When the top level *proof* or *disproof* subgoal has an answer, the question itself is answered - YES if the *proof* attempt was successful; NO if the *disproof* attempt was.

Unless we had a YES (or NO - in which case this subgoal is ignored) evaluation, we now enter the new subgoal through the subgoal splitting process, and add access actions for it. The subgoal is also added as a child node of the parent subgoal it started from. The subgoal tree is not finished after the splitting process - inference can add to it through the subgoal actions. However, if this subgoal has already been created from a previous action, or the parent subgoal already has an identical child subgoal, the subgoal

will not be added.

4.3.4 Answer Combinations

To combine several answers for a YES/NO question, the following is done: (Note that the rules and the process is identical to that used by the adjudication in formula assertion). All positive (YES) answers are gathered, and the resulting probability determined using the rule

$$(1 - (1 - p_1)(1 - p_2) \ldots (1 - p_n))$$

where the p_i are the probabilities of the different answers. All negative (NO) answers are gathered and combined in the same way. Order is not important when combining supporting answers.

If there were both positive and negative answers, the resulting probabilities of these are now combined, using the rules

$$p_yes = (p_yes - (p_yes * p_no))/(1 - (p_yes * p_no))$$

$$p_no = (p_no - (p_yes * p_no))/(1 - (p_yes * p_no))$$

where the p_yes is the resulting probability from combining all the positive answers, and the p_no is the probability from combining all the negative answers. The answer is YES is the resulting p_yes is higher than p_no, NO otherwise. Note that combinations of negative and positive using these rules are order dependent - this gathering of like answers and combination using the non-order-dependent rule counteracts that for answer combination, but all answers are not available during forward inference so another rule will have to be determined that isn't order dependent to use there.

Note: the system is not yet sophisticated enough to determine inference path subsumption, or that one path uses more specific information, so occasionally answers will be combined where one should have been discarded as redundant (e.g. two identical paths except that one uses *many humans are happy* while another uses *most girls are happy* - the *girl* rule is more specific than the *human* rule so that should be used RATHER than the other one, not in combination with it).

The combination method is not being used on WH questions yet.

4.3.5 Comments on Question Answering

When using the question answering subsystem, there are some things to be aware of. There is no differentiation yet between input-driven rules and explanation (goal-directed) rules, so the same ones are used for both. Although you can restrict some rules to only be active during this process, the ones that are used for input-driven inferencing are available here too. For some questions one might ask, the answer will already be there through input-driven inferencing. The question subsystem can help you for questions such as "Does there exist a ..." or "Do all ..." or for specific questions that input-driven inferencing would normally have answered already but was stopped due to low probability or too many rule applications, or for questions whose answers involve the rules which have been flagged as *goal only* . If additional rules have been added after the story information has been loaded (without ***rule-forward*** on), the question answerer can help there too.

4.4 WH Questions

A wh-question (to this system) is a quantified formula whose top level quantifier is *WH* . The question answering mechanism described above is used to answer the question, with one difference - it doesn't stop

after the first answer is obtained - it continues looking for more YES answers. The user may specify that all answers are to be found, or only n answers. This number parameter can be useful if you only want a few answers - for example, if one needs to find out who knows Lisp to ask him a question, only a few names are needed, but for a classlist all names would be.

For example,

(q '(WH x (x girl)))
would give a list of all the girls the system knows about.

(q '(WH x (E y ((lrrh meet x) @ y))))
will show everyone who met Little Red Riding Hood.

(q '(WH x (x human) (x know-about LISP)) 2)
will give the first two people it finds who know about Lisp.

The answers returned for a wh-question contain lists of the entities which matched the variables quantified by WH in the question. If any of the matches is a functional term, the system will try to simplify it using a specialist. Note that answer adjudication is not being done yet for wh-questions.

Currently wh-questions are handled by the main question answering mechanism. The user may specify if all answers are to be returned or only the first n . Only YES answers are considered. A wh-question is one which contains a variable quantified by WH, which is not embedded in a term somewhere. While yes/no questions are answered with YES or NO, wh-questions return the list of entities which match the variables quantified by WH in the question. However, if there cannot be any entities of that type, the system will answer NO (just as if the question were phrased with E instead of WH). The same answer may be obtained through different inference paths. The system will detect these, but only when the path has finished and it can compare the matching entities.

4.5 Saving Question Results

Optionally, the results of a query can be permanently saved by the system. If the ***save-results*** flag is t , any query result which required inference is saved (either a specialist answered it, or one or more rules had to be applied to answer the question). If the question was existentially quantified, or was a wh-question, the "answers" are substituted for the variables before the result is saved. The user may control how hard the system had to work to get the answer before it is saved - either all results are saved, or only those requiring n or more rules to answer the question (flag ***result-difficulty***). Optionally, input-driven inference will be done on the result when it is loaded (***result-forward***).

Note that if the answer is a combined answer (i.e. several answers were found and adjudicated), further calculations using the formula may result in inaccurate probabilities. When this is solved for input driven inference, this will be remedied as well.

4.6 Controlling the Question Answerer

Parameters which may be tweaked to control the proof process:

General Question Parameters:

qa-iterations

The maximum number of actions to be done from the agenda during a question answering attempt. If the attempt stops due to reaching this maximum, it can be restarted by just doing a *(q)* command with no formula. The default is 10, but it may be changed using the *tweak* command.

question-effort
The default effort level which will be used for all questions. The values are 0 -lookup/specialist evaluation only, 1 - allow subgoal splitting, 2 - allow inference, 3 - allow assumptions. The default is 3 - the maximum effort level.

question-threshold
The probability threshold above which answers are accepted. Any answer obtained with a probability lower than this is rejected (but is saved on the *other-answers* list). This threshold is initially 0.4, but may be reset using *tweak* .

minimum-effort
The minimum number of iterations to use in finding answers to a question. If an answer with probability less than 1 is found, and this many iterations have not been used yet, the system will try again to find another answer. The default is 5.

max-wh-difference
The maximum difference allowed between the depth of the first answer obtained for a wh-question, and any subsequent answer. If set to nil, all possibilities are tried to exhaustion. The default is 0, which gives the most efficient result, although may not give all the desirable answers in all cases.

Parameters which affect position in the question-answering agenda:

qa-access-weight
This is the penalty used for access actions when ranking them to put them on the agenda. It is initially 40, which ensures that subgoals get preference over access actions. This may be changed using *tweak* as well.

contra-weight
This is the penalty for contrapositive subgoals, and for accesses which will lead to contrapositive subgoals. It is initially 10. This may be changed using *tweak* as well.

rank-importance
This is the weight of the rank when calculating agenda position for actions. It determines how much higher ranked actions are preferred over lower ranked actions. It is initially set to 100, which makes it the most important features for ranking agenda items, but may be reset with *tweak* .

qa-depth
This is the "maximum" depth that will be considered in the agenda positioning calculation. When a subgoal's depth gets to half this amount, the amount it contributes to the agenda position gradually tapers off until at the maximum, there is no contribution. This makes the subgoal or access action quite undesirable, unless it is the only one available. This is initially set to 20, which means that the rank importance starts tapering off at 10.

prob-importance
This is the weight of probability when calculating agenda position for actions. It determines how much actions with higher probability are preferred over those with lower probability. It is initially set to 100, which makes it the second most important feature forranking agenda items (probability numbers are less than 1, rank numbers are usually greater - this is why probability becomes second

most important feature even though the "importance" numbers are the same), but may be reset with *tweak* .

initial-prob

This is the probability to use in calculating the agenda position of initial (depth 0) subgoals. It is set to .75 initially, and should be less than 1 to prevent the system from preferring the initial accesses to later ones which have actually retrieved information from the knowledge base. Note: this does NOT affect the probability of the answer, only the agenda positioning of initial subgoals.

interest-importance

This is the weight of the subgoal's interest to complexity ratio when calculating agenda position for accesses or subgoals. It determines how much actions with higher interest measures are preferred to those with lower measures. It is initially set to 10.

difficulty-importance

This is the weight of subgoal difficulty when calculating agenda position for accesses or subgoals. It determines how much actions with lower difficulty measures are preferred overthose with higher measures. It is initially set to 50.

quantified-difficulty

This is the difficulty value added to subgoals which are quantified but are not conditionals (e.g. existentially quantified). It is set to 20, which penalizes existentially quantified subgoals moreso than simpler subgoals, but less than the more difficult subgoals requiring assumption.

conditional-difficulty

This is the difficulty value added to conditional subgoals which are solved by assuming the antecedent and trying to prove the consequent. It is initially set to 30 (quite high).

split-difficulty

This is the difficulty value added to subgoals which are split into several simpler subgoals, without any assumptions. It is initially set to 10 (low).

assume-difficulty

This is the difficulty value added to disjunctive subgoals which are solved by assuming the negation of one of the disjuncts and trying to prove the rest. It is initially set to 30 (quite high).

mp-weight

This indicates how much to penalize access actions which are looking for mp's to infer with. It is initially set to 10000, a very large penalty, as the mp's are rarely helpful in question answering (but can be, so they must be available).

residue-penalty

This indicates how much to penalize subgoal actions which have a residue involved in the comparison. The residues are rarely helpful and so the penalty is set quite large (10000).

use-inherit

This flag indicates whether or not to consider "inherited" interest in positioning agenda items. It is set to *t* (use) by default.

qa-inherit-amount

This flag controls how much the inherited interest descreases from parent subgoal to child subgoal. It is initially set at 20.

favor-interest

This is the amount to "prime" an agenda item with when it is moved to the top of the agenda after too much time has been spent on the other proof/disproof attempt. It is initially set to 1000. Note that ***use-inherit*** must be t for this to be considered, as the interest is added as an inherited interest value.

favor-position

This is the amount above the highest agenda item to place an agenda item when it is moved to the top of the agenda after too much time has been spent on the other proof/disproof attempt. It is initially set to 100.

Parameters which affect ACCESS actions:

max-wffs

The maximum number of retrieved formulas to test in an access action. Any formulas not tested are put back on the agenda, so nothing is lost. This is used to control how long actions can take. It is initially set to 10, but may be changed using *tweak* . Changing it to a higher value may make some accesses take longer because more wffs are tested, but the matching wff may be found sooner. There is a trade-off - if this access action does not contain the wff we need, having a low *max-wffs* allows us to get on to the next action more quickly.

max-class

This contains the maximum number of classifications to retrieve wffs for in an access action. Additional classifications are put back on the agenda for a later time. This is initally set to 5, and may be reset using *tweak* . If this parameter is set to 1, and *max-wffs* set to a very large number, the access actions are more similar to **ECoNet** 's.

unify-sorts

indicates wheterh sorts should be considered when unifying terms. Initially set on (t).

Parameters which affect SUBGOAL actions:

goal-forward

This flag indicates whether or not input-driven inferencing should be attempted on assumptions made during the proof process (it will only have an effect if input-driven inferencing in general is turned on - using ***story-forward*** and **rule-forward**). It is initially turned off, but may be turned on using the **tweak** command.

Trace values which can be used to view the question answering process:

subgoal -Shows splitting of subgoals

qa-test -Shows tests for (in)compatibility between set-of support clauses and accessed clauses

qa-success -Shows successful subgoal actions during the qa process

qa-access -Shows access actions done during the qa process

qa-eval -Shows evaluations and simplifications that take place during the qa process

qa-time - Keeps track of how much time each question takes

qa-iterations - Keeps track of how many iterations each question takes.

qa-answer - Displays the answers to the question

qa-after - Prints successful subgoal (proof trace) AFTER question has been answered.

qa - Traces the minimum essentials for question answering - the time and iterations, and the answer

qa-all - Traces everything during qa, including *qa-time, qa-success, qa-access, qa-eval, qa-test, subgoal*

qa-int - Traces the minimum to see how the question was answered, including *qa-success, qa-answer, subgoal, qa-time, qa-iterations* .

4.7 TroubleShooting

Sometimes a question does not get answered the way you think it should. This section describes what to do to find out why. If after trying these things you still can't figure out why the question is not being answered correctly, don't hesitate to contact the authors.

4.7.1 What to do if an Answerable Question is NOT Answered

The first thing to do is to ensure that the information needed to answer the question is actually there first - both rules and facts. Once you've figured out what you think should happen, you can compare it against what the system is actually doing to get some idea of what is going wrong.

To see what is happening, try tracing *qa-int* and *qa-test* , and ask the question *(q *question*)* again to see what is happening. You want to see if it retrieves the rules and facts you expect it to use to solve the problem, as well as seeing if it compares the question subgoals successfully with those formulas. You should probably set ***qa-iterations*** to something small (like 1) so you are not overwhelmed with output, and use **(q)** to continue the question after it stops. You can also look at the agenda (*(display 'agenda)*) between steps to see what the next actions are going to be. Common problems which can be detected this way are described shortly.

Another thing to try is to force the system to spend all its resources trying to answer the question the way you think it should be. If you think the answer should be yes, use the command **proof-q** instead of **q** ; if you think it should be no, use the command **disproof-q** . If this gives an answer, then the problem lies in the ordering of the agenda items. It is possible that changing the interest level of a predicate might help, but more likely it won't. You can play with the parameters controlling agenda positioning, but this most often leads to frustration, with one set of parameters working well for one question, but no others, etc. This is a problem best left to the authors.

Some possible problems are:

A rule or fact which is needed is not found
This could be a classification problem in that the appropriate classifications are not checked under to find the rule. Ensure that the types given in the rule and question are not incompatible. There also could be a problem in that a particular key of a rule or question which should be a trigger key wasn't selected as such. The same problem can occur during input-driven inferencing, and the troubleshooting section of

that chapter can give some assistance as to what can go wrong and how to fix it. To trace information on classifications used in question answering, trace *qa-access* .

The comparison between subgoal and retrieved wff failed
A common problem is to ask a question using **, and have facts which only use *. It is usually better to ask the weaker form of the question. Trigger keys may not have been selected properly, in which case the system won't try to compare the right keys in the subgoal and retrieved formula. This is analagous to the input-driven inference problem of unsuccessful matches; see that section for more hints. Specialists may be required to compare two keys, and if this fails and you think it shouldn't have, trace the particular specialist to see what has happened. A common problem here is that information applicable to the specialist wasn't actually entered into the specialist because of missing sorts on the arguments.

An evaluation failed
At some point along the inference path, evaluations may be needed, and these may be done by simple lookup or by specialists. If a subgoal is entered into the system which seems like it could be simpler, or could be completely answered, try tracing the specialist that you believe should have helped with it. There may be a sort problem like that described in input-driven inference (missing sorts on input cause facts to not be entered into the appropriate specialists), or perhaps the specialist can indicate why something won't evaluate (trace the specialist's actions). Once you have the problem narrowed down to something like this, try evaluating that piece of information outside the question (by asking it as a question) - at least the problem has been simplified to a simpler question which doesn' t seem solvable.

Type information is missing
A common problem in answering wh-questions or "does there exist" questions is that the entity which the question was intended to find cannot be because no type has been associated with it. These types of questions require that the system start from the top level *entity* type, and work down the type hierarchies, looking for instances of each subtype to use. If the desirable entity was not given a type, it cannot be found this way. All entities should be given types on input.

The question itself should also be phrased with type information (if available), mostly for efficiency reasons. When asking wh-questions or "does there exist" questions, if a type is not given for at least one of the variables, the system will have to waste much of its resources looking at every possible entity in the system. If given, types can narrow this search down dramatically, and this can make the difference between the question being answered and not answered.

Not enough resources were devoted to the question
The number of iterations allowed was not high enough (unlikely if the default was used), or the effort level allowed to be used by the specialists was not high enough (again unlikely if the default is used). It is possible for some questions to require more than the normal maximum of iterations, but this should only occur for some wh-questions. Asking with **proof-q** or **disproof-q** should help indicate if this is the case. If the appropriate question routine answers the question, then perhaps increasing *qa-iterations* will enable the question to be answered by **q** .

The system is running off on a tangent
It is quite difficult to order and control the agenda for optimum performance on all questions. Controls have been put in to try to prevent this, but occasionally a question will be asked for which the usual

ordering parameters do not work well. Playing with these can be quite a hassle. Check the rules involved first to make sure they make sense and all the bracketing is correct, and then contact the authors.

4.7.2 What to do if a Question is Answered Wrong

A scarier problem occurs when a question is answered incorrectly. In this case, a NO or YES answer is recieved for a question which should have been answered the other way, or should be unanswerable. In this case, look at the inference path used to get the answer (use **(display 'subgoal)**), and check to see that all the rules and facts are being applied correctly. There may be an error in a rule which causes this. If nothing strange can be detected in this way, contact the authors.

Chapter 5

Specialists

Specialists are special purpose inference mechanisms designed to accelerate certain parts of the theorem prover's operation in certain domains. These are domains where the usual inference rules are either grossly inefficient, or cannot be used at all. These specialists may use different representations (trees, graphs, etc) and methods to achieve their efficiency. Some examples of domains where special methods can help are time, color, motion, symbolic expressions, space, part-of relationships, etc.

The specialists may assist the theroem prover in several operations, including literal evaluation, function evaluation (simplification), and comparison of literals and predicates. They also do checkpointing and retraction so that their representations remain consistent with the rest of the system.

There are two types of specialists in EPILOG: those which maintain their own representation of story facts in their domain, and those which only contain static information on predicates in their domain. The specialists can be used to evaluate a wff (tell whether it is true or false), to simplify a wff by evaluating functional terms, and even to help compare two wffs. Asserted wffs are also offered to the specialists in case they want to maintain the information in their own representation. Note that only formulas with probability 1 are entered into specialists' representations, and all evaluations done by specialists have probability 1.

The current system has a number of specialists available for doing inference with temporal relations between episodes and time (*time-specialist*), arithmetic relations and functions with numbers (*number-specialist*), equality with constants (*equality-specialist*), set membership relations (*set-specialist*), relations and functions with strings (*string-specialist*), relations between types (*type-specialist*), episodic relations (*episode-specialist*), non-type predicates arranged in a hierarchy (*hier-specialist*), relations between colors (*color-specialist*), and part hierarchies and part-of relationships (*part-specialist*). There is also a specialist for handling the "meta" information which is often required for meaning postulates. A specialist called the *other-specialist* enables easy addition of external routines to the system.

For example, if we assert

(e1 before e2)

the *time-specialist* would be able to take that and store it in its graphical representation. Then later, if we asked

(e2 after-1 e1)

(where *after-1* means strictly after), the time specialist would be able to answer No, which would save numerous applications of rules about temporal relations. Also, a formula like

(n1 equal (add 7 8))

could be simplified to

(n1 equal 15)

by the *number-specialist* .

When the system first starts up, the following specialists are automatically active: *type-specialist* , *episode-specialist* , *hier-specialist* , *part-specialist* , *meta-specialist* , and *other-specialist* . The others must be activated in order to use them - using the command **use-spec** (e.g. *(use-spec 'time-specialist 'number-specialist)*).

Note that specialists currently do NOT handle probabilities in their domains, so will only be allowed to store formulas with probabilies equal to 1 in their own representations. Also, any evaluation made by a specialist has probability 1.

The next sections in this chapter describe the specialist interface and using specialists in general, followed by a description of each individual specialist, what it does, how it does it, and how to control it.

5.1 Using Specialists

These functions show how to see what specialists are available or active, and how to activate one. More information on individual specialists can be found in the chapter "Specialists".

(use-spec *specialists*) [*function*]

Purpose: Activates specialists, loading in their object code, and putting information on the property lists of affected predicates and functions. If information in a specialist's domain is to be entered, and it is deirable that the specialist assist the system with that, then the specialist must be activated first, using this command.

Syntax: *specialists* are the names of the specialists to be activated. Specialists can be indicated by their full names or specified nicknames. If no specialists are indicated, the ones remaining available that have not yet been activated will be displayed.

Examples:

(use-spec 'time-specialist 'number-specialist)

Remarks: Only specialists from the **available-specialists** list (use command *(display 'available-specialists)*) may be activated. A message will be printed if a specialist has already been activated. Some specialists are automatically activated when the system starts up. Nicknames are accepted for some of the specialists - for example, the *time-specialist* may be started with *(use-spec 'time)* .

To determine what specialists are active or available, or to examine what communication paths have been set up by the specialists, the *display* command can be used with the following "topics":

(display 'active-specs)
Displays the specialists that are currently active.

(display 'avail-specs)
Displays the specialists that are available for activation. Note that this includes the already active ones.

(display 'spec-effort)
Displays the current effort levels for entry and evaluation.

(display 'interested-party *concept* **)**
Display the specialists and literals on the interested party list for *concept* . If *concept* is a list of concepts, the interested party list for each is displayed. Interested party lists are an important part of communication between specialists, and will be discussed shortly.

5.1.1 Controlling the Specialist Interface

Controls for the specialist interface include several flags:

spec-enter
If t, input and inferred formulas will be sent to interested specialists to store in their own domain. If nil, no additional storage is done. The default is t.

spec-evaluate
If t, formulas will be sent to interested specialists for evaluation when necessary. If nil, specialists cannot assist in literal evaluation. The default is t.

spec-compare-preds
must be t for specialists to be used in comparing predicates. Note - this flag should NOT be tweaked to nil except for certain kinds of testing.

spec-compare-lits
If t, specialists may assist in comparing literals during question answering. The default is nil, as this can be an expensive operation and its uses are limited. Some complex time questions may require this comparison, but determining when it is useful is still an ongoing problem.

fwd-spec-compare
If t, specialists may assist in comparing literals during input driven inferencing. This is rarely helpful, but interesting to play with, so the default is nil.

spec-assert
If t, specialists may make assertions back to EPILOG which are then entered in the same way as other inferences.

Trace values which apply to all the specialists are as follows:

spec-test - Traces comparisons between literals handed to the interface.

spec-entry - shows literals handed to the interface for entry into

spec-eval - show literals handed to the interface for specialists to evaluate

interested-party - Traces addition to interested party lists, and reassertion of literals from there.

function-eval - Traces evaluation of functional arguments by the specialists.

spec-int - traces all aspects of the specialist interface, including *spec-test, spec-entry, spec-eval, interested-party,* and *function-eval* .

5.2 Details of the Specialist Interface

In this system, specialists are allowed to assist the theorem prover in several ways:

1) Literal evaluation
A simple literal may be quickly evaluable by a specialist where the main theorem prover may need many inference steps to do it. For example, if we have

(t1 before t2)

(t2 before t3)

and

(t3 before t4)

in our knowledge base, the time specialist can evaluate

(t1 before t4)

in one step, whereas the theorem prover needs two inference steps using the transitivity axiom.

2) Function evaluation
Functional terms may be simplified by a specialist, simplifying the overall proposition. For example, *(max-of n)* might simplify to *3* . This is used to simplify terms. In addition, asserted literals with functional terms can be transformed into several literals which set the desired information in the specialist (functional terms are not normally sent to the specialist on assertion). For example, if

((cardinality-of s1) less-than 30)

is asserted, this will be transformed into several assertions (to the specialists only!)

(d_number less-than 30)

(s1_set has-cardinality d_number)

This allows for complex relations among terms expressed as functions to be maintained (e.g. *((duration-of e1) less-than (duration-of e2))*).

3) Predicate comparison
Specialists which do not maintain their own independent storage of story facts "know" the relationships between particular predicates. These predicate relationships can be used by the main system to determine incompatiblity or compatibility of literals in a single step which might otherwise take several inference steps. This is used for simple lookups of information in the knowledge base (for example, this is how the system knows that

*((a kiss b) * e1)*

is true when it only has

*((a kiss b) ** e1)*

or that LRRH is a human when we have *(LRRH girl)* stored. This is also used during backchaining to determine compatibility of the clause we have and one we want to use to backchain with, and during proof by contradiction to determine that the set-of-support clause we have is incompatible with a proposition we retrieved from the knowledge base.

4) Literal comparison
Specialists who maintain their own individual storage of story facts can sometimes determine incompatiblity or compatibility of literals which do not conform to the usual rules (same predicate, appropriate signs - same for compatible, different for incompatible). These specialists usually work

with literals which have more than one argument (e.g. time, numbers) and so the predicate comparisons alone are not enough. Also, the internal representation may be used which then also includes other story facts, so we can combine a number of inference steps into one. These comparisons can help the theorem prover in proof by contradiction, backchaining and input-driven inference.

5) Assertion
Although a specialist doesn't really "help" with assertions, assertions in a specialists domain (with probability = 1) should be sent to it to maintain in its own representation, for later use in literal evaluation, function evaluation, and literal comparison.

5.2.1 Specialist Entry and Evaluation

The specialist interface decides which specialist(s) would be interested in a propostion being asserted (entered), able to evaluate a functional term or a literal, or able to tell if two literals or predicates are incompatible or compatible. Determining when a specialist is appropriate to call could be done using pattern matching, but this can be computationally expensive. A simpler mechanism is used which doesn't guarantee that all the arguments are appropriate for the given specialist. The specialist itself can do this last minute checking much more efficiently than the general interface, as it can make generalizations.

Each predicate and function has a list of specialists associated with it. Applicable specialists for function evaluation, literal evaluation and assertions are simply those specialists on the predicate's or function's property list. For comparing predicates or literals, the list of applicable specialists is the intersection of the two lists of specialists from the predicates involved.

In addition, each specialist has a list of "allowed first argument sorts" which gives an additional check for literal evaluations and assertions to be sent to a specialist. If the first argument matches one of these sorts, the specialist will be invoked, and further argument checks are done by the specialist itself. Literal comparison does not check this first argument sort; the specialist must still do unification, and until that is complete, the first argument sorts may not be known. Functions sorts may be completely different than those used by literals in the specialist's domain (for example, the time specialist function **date** uses *number* arguments), so this is left up to the specialist to test as well.

For entry and evaluation, all terms are evaluated first (so far only functions with constant terms are evaluated). The literal with the evaluated arguments is then entered or evaluated by the specialists. In **ECoNet** , for assertions, the specialist interface was called to evaluate literals in the hopes of simplifying the proposition (including replacing complex functional terms with their values after function evaluation). The simplified proposition is then kept by the system.

When evaluating, some specialists have different levels of effort that can be used to find an answer. For example, the time specialist has two: constant time only operations (effort 0), and full blown search (effort greater than 0). The specialist interface has tweakable global effort values which it gives the specialists to use - one for entry ***specialist-entry-effort*** (for any evaluation done for consistency testing for assertions), and another for evaluation ***specialist-eval-effort*** (function evaluation and question answering). These are initially set to the maximum effort level, but may be modified by the user using the **tweak** command. The evaluation effort is also used when determining if two literals are incompatible or compatible.

All applicable specialists are called for assertions, and for determining literal incompatibility (we save all results and try them one by one). For literal evaluation, function evaluation, and determining predicate incompatibility, we continue calling specialists until one returns an answer (other than *unknown* or *nil)* .

When information is modally embedded, information is entered (asserted) into ALL equivalent subnets within the specialist. Evaluation and literal comparison only required that one subnet be examined though.

Functions which have corresponding relational predicates may be "flattened" (e.g. ((cardinality-of s1) gt 3) -> (s1 has-cardinality cardinality-ofs1) & (cardinality-ofs1 gt 3)) to make it easier for the specialists to work. This is done only on assertion, and all of the resulting conjunction wffs are asserted (to the specialists only).

5.2.2 Specialist Communication

Specialists may also communicate with each other through the specialist interface. There are two types of such communication: *immediate evaluation* , and *delayed communication* . Immediate evaluation is done when a specialist requests that a function or literal be evaluated. All communication is channeled through the interface itself, so when a specialist requests an evaluation, it is passed to the interface, which decides who to send it to in exactly the same manner as if it were assisting the theorem prover (as described above).

Delayed communication involves one specialist adding something to a concept's interested party list. The interested party list for a concept consists of entries containing the specialist interested, the assertion that made him interested, and the current subnet context. Further assertions about that concept cause the literals on the interested party list to be reasserted to the specialists interested with them. A specialist may also inform the interface that something has changed about a concept not directly involved in the current assertion (via propagation for example). The interested party list for this concept will then be reasserted after the current literal assertion is finished.

Specialists may be activated using the **use-spec** command. In addition, the user may select what operations specialists are allowed to assist with, so that a specialists need not participate in all possible operations even if it is activated. There are several flags set up: ***spec-enter*** (must be true to enter anything in a specialists domain), ***spec-evaluate*** (must be true for specialists to evaluate literals), ***spec-compare-preds*** (must be true for specialists to be used in comparing predicates - Note - this flag should not be tweaked off except for certain kinds of testing), and ***spec-compare-lits*** (must be true for specialists to be used in comparing two literals). The last flag can be tweaked off for most normal operations.

5.2.3 Specialist Subnets

When storing information into the different specialist representations, subnets are taken into account. In **EcoNet** , modal propositions were of the form

[john believes [e1_episode before e2_episode]]

[john believes [mary believes [e1_episode before e2_episode]]]

where EPILOG handles them somewhat differently

*[E x [[e1_episode before e2_episode] _ x] [E y [[john believe x] ** y]]]*

[john believes (that [e1_episode before e2_episode])]

*[[[e1_episode before e2_episode] _ p1] and [[john tell mary p1] ** e3]]*

[john believes [mary believes (that [e1_episode before e2_episode])]]

In ECoNet , the subnet string was just the first arguments of each embedded modal combined with /'s

between (e.g. */john* , */john/mary*). For **EPILOG** , however, things aren't quite that easy. Because of the storage technique of storing within the subnets based on the content of the proposition, we already have an ordering constraint on modal propositions forcing the one describing the content to be first (the one with the _), and the ones about who believes it to come after. This means that when we try to enter something like

[[e1_episode before e2_episode] _ p1]

we will not have any context available for *p1* yet. So we have to wait for propositions of the form

*[[john believe p1] * ep1]*

and then strip off the episodic embedding clause, detect the modal context and use it to build a subnet string (*/john-believe* for the first two examples, */johnmary-tell* for the third "tell" example, and */john-believe/mary-believe* for the last example). Then the content of the propositional argument (the last) must be obtained by looking it up in the knowledge base under that argument and topic *tp.content* . The the embedding _ part is stripped off the first wff found there (note, this assumes that there will only be one phrase describing the content of a propositional constant), The remaining literal is the one we actually want to store in the specialist representation, and the subnet context is given by the one we started with.

The same technique is used to separate the literal to be evaluated from the subnet context. This doesn't work very well if the propositional argument is a variable, however, since classifications are not prepared for variables (so there is nothing under *tp.content* for *propos-x* for example).

5.3 Type Specialist

The domain of the type specialist consists of literals which contain type predicates. A type predicate is defined as a predicate which is on one of the type hierarchies, which are set up using command **add-hier** . No sorts are involved here, and no functions available either. Predicate operators are not handled by this specialists - meaning postulates are required to make an assertion from them that the specialist could use.

5.3.1 Using the Type Specialist

This specialist (**'type-specialist**) is automatically started up when **EPILOG** starts up, so is always active. The hierarchies may be printed using command **print-hier** , or using the following display command.

(display 'hier *hier-node* **)**

Displays the hierarchy below *hier-node* , or all hierarchies if no *hier-node* is specified. If *'-full* is specified, the subtypes are recursively displayed, with their preorder numbers. Otherwise only the hierarchy types and connections are printed.

Some hierarchies are set up automatically by the system - *entity* (an overlap hierarchy), *term* (contains the sorts), and *meta-entity* (which contains the meta "sorts" - wff, etc). All type hierarchies are automatically connected to *entity* .

The hierarchies themselves may be changed in order to control the actions of the specialist. Consistency testing may optionally be done - controlled by ***specialist-entry-effort*** . The flag ***spec-compare-preds*** controls whether or not the specialist gets invoked at all, and the flag ***specialist-eval-effort*** controls how hard it tries to get an answer. An effort level of 0 means constant time checks

only (in same hierarchy), 1 means two hierarchies may be involved, and higher means a full search using all relevant connections will be done if necessary.

Trace values for the type specialist are:

type-test -.SHows comparisons between type predicates

5.3.2 Details of the Type Specialist

Currently, this specialist uses the type hierarchies as a logically true representation of the relationships among the predicates that appear in the hierarchy. If any change has been made to the hierarchy since the last renumbering, or there aren't numbers on two type predicates it is trying to compare, the hierarchies are renumbered. This numbering makes possible constant time determiantions of the relationships betwen predicates.

For example, in the "thing" hierarchy shown below, *thing* is subdivided into *physical-object* and *abstract-object* , which are further subdivided - *physical-object* into *living-thing* and *non-living-thing* , and so on. The hierarchy can be used to determine that *wolf* and *human* are disjoint because there is no overlap between the numbering range associated with *wolf* [38,38] and *human* [17,26]. *Wolf* is subsumed by *creature* because the numbering range associated with *wolf* [38,38] is within the numbering range associated with *creature* [16,40].

```
THING [1,112]
|      PHYSICAL-OBJECT [2,108]
|      |      LIVING-THING [3,40]
|      |      |      PLANT [4,15]
|      |      |        ...
|      |      |      CREATURE [16,40]
|      |      |      |      HUMAN [17,26]
|      |      |      |      |      ADULT [18,20]
|      |      |      |      |      |      MAN [19,19]
|      |      |      |      |      |      WOMAN [20,20]
|      |      |      |      |      MINOR [21,26]
|      |      |      |      |      |      ADOLESCENT [22,22]
|      |      |      |      |      |      CHILD [23,25]
|      |      |      |      |      |      |      GIRL [24,24]
|      |      |      |      |      |      |      BOY [25,25]
|      |      |      |      |      |      INFANT [26,26]
|      |      |      |      ANIMAL [27,40]
|      |      |      |      |      MICROBE [28,28]
|      |      |      |      |      BUG [29,31]
|      |      |      |      |      |      ...
|      |      |      |      |      LARGER-ANIMAL [32,40]
|      |      |      |      |      |      FISHLIKE-ANIMAL [33,33]
|      |      |      |      |      |      BIRDLIKE-ANIMAL [34,35]
|      |      |      |      |      |      |      BIRD [35,35]
|      |      |      |      |      |      REPTILE [36,36]
|      |      |      |      |      |      WARM-BLOODED-QUADRAPED [37,39]
|      |      |      |      |      |      |      WOLF [38,38]
|      |      |      |      |      |      |      FOX [39,39]
|      |      |      |      |      |      SIMIAN [40,40]
|      |      NON-LIVING-THING [41,108]
```

```
|      |      |      INANIMATE-NATURAL-OBJECT [42,55]
|      |      |      |      ASTRONOMICAL-OBJECT [43,43]
|      |      |      |      ROCK [44,44]
|      |      |      |      MOUNTAIN [45,45]
|      |      |      |      LAKE [46,46]
|      |      |      |      ...
|      |      |      ARTIFACT [56,107]
|      |      |      |      DEVICE [57,77]
|      |      |      |      |      VEHICLE [58,60]
|      |      |      |      |      |      CAR [59,59]
|      |      |      |      |      |      TRUCK [60,60]
|      |      |      |      |      ...
|      |      |      |      FURNITURE [78,82]
|      |      |      |      |      BED [79,79]
|      |      |      |      |      TABLE [80,80]
|      |      |      |      |      ...
|      |      |      |      ...
|      |      |      FOOD [108,108]
|      ABSTRACT-OBJECT [109,112]
|      |      THOUGHT [110,110]
|      |      IDEA [111,111]
|      |      GROUP [112,112]
```

The *thing* hierarchy is an **exclusion** hierarchy - all sibling nodes are mutually exclusive. **Overlap** hierarchies are also allowed - in those subsumption can be determined, but not disjointedness. For example, the subtypes of the *explosive* hierarchy below are not necessarily incompatible.

```
EXPLOSIVE [1,]
|      BOMB [2,4]
|      |      CARBOMB [3,3]
|      |      TIME-BOMB [4,4]
|      OTHER-EXPLOSIVE [5,6]
|      |      CHEMICAL-MIX [6,6]
```

Any number of different hierarchies are allowed. They will all be connected at the root level to the *entity* hierarchy so that individuals of any type can be located even when no specific type information is given. For type hierarchies which are "tangled" - that is, the same predicate appears in more than one hierarchy, we can use this connection to help determine relations across hierarchies. This works much like the "metagraph" in the time specialist - within each hierarchy, the preorder numbering scheme is used, and between hierarchies the connections are used. For example, if we add another breakdown of human:

```
HUMAN [1,4]
|      CAUCASIAN [2,2]
|      NEGRO [3,3]
|      ASIAN [4,4]
```

We can determine that *Caucasian* is subsumed by *creature* because *Caucasian* is subsumed by *human* (using the preorder numbering from the new hierarchy), and *human* is subsumed by *creature* (using the

preorder numbering in our original *thing* hierarchy). Any number of connections between hierarchies may be used. Consistency checking on input of these hierarchies is very important to ensure that there are no loops though! (The system does this automatically for you unless the ***spec-entry-effort*** flag has been set to 0.)

The type hierarchies have an additional role to play in **EPILOG** - they help to organize the formulas, and when the system searches for rules to apply to an input formula or question subgoal, it will "climb" the type hierarchy to find additional formulas (e.g. given that *wolf1* is a *wolf* , it would look for rules about *wolf1* , then rules about *wolves* , then rules about *warm-blooded-quadrupeds* , and so on).

The type specialist applies the same information and inference methods in modally embedded contexts (both subnets and simulation environments) as it does to top-level literals. If the system has been informed that *wolf* is a subtype of *animal*, then not only can it prove *(wolf1 animal)* from *(wolf1 wolf)*, but it can also prove *(lrrh say (that (wolf1 animal)))* from *(lrrh say (that (wolf1 wolf)))*, and *(lrrh believe (that (wolf1 animal)))* from *(lrrh believe (that (wolf1 wolf)))*.

5.4 Predicate Hierarchy Specialist

The domain of the predicate hierarchy specialist consists of literals which contain predicates that a hierarchy has been built for. These predicates are set up using command **add-hier** . Note that before anything is added to one of these hierarchies using **add-hier** , the command **(set-hier-type** *newhier* **'pred-hier)** must be issued, or it will be treated as a type hierarchy! No sorts are involved here, and no functions available either. Predicate operators are not handled by this specialist - meaning postulates would be needed to infer the appropriate thing.

5.4.1 Using the Predicate Hierarchy Specialist

This specialist (**'hier-specialist**) is automatically started up when **EPILOG** is loaded, so is always active. Note - all predicates on the same hierarchy must have the same topic indicators for this to work in a natural way. The hierarchies may be printed using command **print-hier** , or using the following display command.

(display 'hier *hier-node* **)**

Displays the hierarchy below *hier-node* , or all hierarchies if no *hier-node* is specified. *'-full* and *'-brief* parameters have no effect here.

The hierarchies themselves may be changed in order to control the actions of the specialist, and consistency checking may optionally be done (controlled by ***specialist-entry-effort*** . The flag ***spec-compare-preds*** determines whether the specialist will even be invoked, and the flag ***specialist-eval-effort*** determines how hard the specialist will work. For an effort level of 0, only constant time checks are done (in same hierarchy), 1 allows two hierarchies to be involved, and anything higher will use a full search using all relevant connections if necessary. of testing.

Trace values for the predicate hierarchy specialist are:

hier-test - Shows comparisons between predicates on the hierarchy

5.4.2 Details of the Predicate Hierarchy Specialist

This specialist uses exactly the same mechanisms (the hierarchies) as the type specialist does. The predicates on the hierarchies are not types, but the hieachy mechanism can help determine subsumption and disjointedness of other kinds of predicates as well. For example, we can have an *occurred-near* hierarchy:

```
OCCURRED-NEAR [1,3]
|       OCCURRED-IN [2,2]
|       OCCURRED-OUTSIDE-OF [3,3]
```

This hierarchy can be used to determine that anything that *occurred-in* something also *occurred-near* it, quickly. The difference between these hierarchies and the type hierarchies described above is that these are not used to help organize the formulas, nor are they "climbed" to find new formulas. They are used only for predicate comparison purposes, which saves on the number of rules which must be entered. They may be tangled just like the type hierarchies.

Like the type specialist, the predicate hierarchy specialist applies the same knowledge and inference methods in modally embedded contexts that it uses on top-level literals (see page 95).

5.5 Part Specialist

The part specialist is used to accelerate question answering and verification during forward inference for literals about relations between parts (although so far it can only handle *part-of* , eventually it will hopefully handle attachment as well). When a proposition is questioned or asserted in **EPILOG** which is in the domain of the part specialist, the specialist is invoked.

5.5.1 Using the Part Specialist

The part specialist is automatically activated when EPILOG starts up.

No special sorts are associated with this specialist.

The domain of the part specialist consists of literals with the following patterns:

Argument1 Sort	**Predicate**	**Argument2 Sort**
	part-of	
	equal	

Literals with predicate *equal* are only handled by this specialist if the entities being equated are parts, or entities about which parts have been asserted. Literals which have a "part" or "part-type" predicate are also examined by the part specialist. A literal like *(h1 hand)* will be examined, and the type of *h1* saved for later use, or evaluated (depending on the desired operation).

Part predicates are those which exist on a part hierarchy, or on a type hierarchy which connects with a part hierarchy. Part predicates can be added using the **add-part-hier** command, and related types using the **add-hier** command. Part hierarchies are assumed to be exhaustive unless otherwise indicated; to indicate otherwise, use the command **set-hier-type** .

5.5.1.1 Part Specialist Display and Control

The part hierarchies may be printed using command **print-hier** , or using the following display command.

(display 'hier *hier-node* **)**

Displays the hierarchy below *hier-node* , or all hierarchies if no *hier-node* is specified. If *'-full* is specified, each subpart, its preorder numbers, and bounds are also printed. Otherwise only hierarchy types and connections are printed.

The usual specialist controls ***spec-enter*** , ***spec-eval*** , ***spec-compare-lits*** work on this specialist. The part specialist has an additional flag ***part-assert*** which controls whether or not the part specialist is allowed to make assertions back to EPILOG. When *(p1 part-of p2)* is asserted, more specific information about *p1* may be known (such as it is a *girl-leg* , and not just a *leg*) and this is asserted if the ***part-assert*** flag is t.

Trace values for the part specialist are:

part-entry - Traces input of part-of relations

part-eval - Traces evaluation of part-of relations

part-assert - Traces assertions made by the part specialist

part-test - Traces literal comparisons made by the part specialist

part-min - Traces interesting things about parts, including *part-entry, part-eval, part-assert, and part-test*

part-all - Traces everything about parts, including *part-entry, part-eval, part-assert, and part-test*

5.5.2 Details of the Part Specialist

There are two parts to the part specialist - one is a set of part hierarchies which are very similar to the type hierarchies (except that subsumption indicates *part-of* rather than *is-a*). The root node of any part hierarchy must be an entity type which occurs on one of the type hierarchies. Any part hierarchies for an entity type are saved with that entity type so that individuals of that type can inherit the appropriate parts. The part and type hierarchies may be tangled (parts are types after all!), but the part hierarchies are not used in organizing formulas, or "climbed" to find new formulas.

```
HUMAN-PARTS   Root: HUMAN   Hierarchy type: EXCLUSION    PART HIERARCHY
        PROPERTIES: (EXHAUSTIVE)    FORM part hierarchy
HUMAN [1,44]   Exactly 1, for a total of exactly 1
|    BODY [21,44]      Exactly 1, for a total of exactly 1
|    |    ARM [36,44]  A set of exactly 2, for a total of exactly 2
|    |    |    WRIST [44,44]   Exactly 1, for a total of exactly 2
|    |    |    ELBOW [43,43]   Exactly 1, for a total of exactly 2
|    |    |    HAND [39,42]    Exactly 1, for a total of exactly 2
|    |    |    |    PALM [41,41]       Exactly 1, for a total of exactly 2
|    |    |    |    FINGER [40,40]     A set of exactly 5, for a total of exactly 10
|    |    |    LOWER-ARM [38,38]       Exactly 1, for a total of exactly 2
|    |    |    UPPER-ARM [37,37]       Exactly 1, for a total of exactly 2
|    |    LEG [27,35]  A set of exactly 2, for a total of exactly 2
|    |    |    ANKLE [35,35]   Exactly 1, for a total of exactly 2
|    |    |    KNEE [34,34]    Exactly 1, for a total of exactly 2
```

```
|    |    |    FOOT [30,33]    Exactly 1, for a total of exactly 2
|    |    |    |    FOOT-OTHER [33,33]         A set of at least 0
|    |    |    |    HEEL [32,32]       Exactly 1, for a total of exactly 2
|    |    |    |    TOE [31,31]        A set of exactly 5, for a total of exactly 10
|    |    |    LOWER-LEG [29,29]       Exactly 1, for a total of exactly 2
|    |    |    THIGH [28,28]   Exactly 1, for a total of exactly 2
|    |    TRUNK [22,26]        Exactly 1, for a total of exactly 1
|    |    |    TRUNK-OTHER [26,26]     A set of at least 0
|    |    |    ABDOMEN [25,25]         Exactly 1, for a total of exactly 1
|    |    |    BACK [24,24]    Exactly 1, for a total of exactly 1
|    |    |    CHEST [23,23]   Exactly 1, for a total of exactly 1
|    NECK [20,20]      Exactly 1, for a total of exactly 1
|    HEAD [2,19]       Exactly 1, for a total of exactly 1
|    |    HEAD-OTHER [19,19]   A set of at least 0
|    |    BRAIN [18,18]        Exactly 1, for a total of exactly 1
|    |    SKULL [17,17]        Exactly 1, for a total of exactly 1
|    |    EAR [16,16]  A set of exactly 2, for a total of exactly 2
|    |    FACE [3,15]  Exactly 1, for a total of exactly 1
|    |    |    FACE-OTHER [15,15]      A set of at least 0
|    |    |    CHIN [14,14]    Exactly 1, for a total of exactly 1
|    |    |    FOREHEAD [13,13]        Exactly 1, for a total of exactly 1
|    |    |    MOUTH [8,12]    Exactly 1, for a total of exactly 1
|    |    |    |    MOUTH-OTHER [12,12]        A set of at least 0
|    |    |    |    TONGUE [11,11]     Exactly 1, for a total of exactly 1
|    |    |    |    TOOTH [10,10]      A set of at least 0 and at most 32, for
|    |    |    |                       a total between 0 and 32
|    |    |    |    LIP [9,9]  A set of exactly 2, for a total of exactly 2
|    |    |    EYEBROW [7,7]   A set of exactly 2, for a total of exactly 2
|    |    |    MOUSTACHE [6,6]         Optional, for a total between 0 and 1
|    |    |    NOSE [5,5]      Exactly 1, for a total of exactly 1
|    |    |    EYE [4,4]       A set of exactly 2, for a total of exactly 2
```

A sample human part hierarchy is given with name HUMAN-PARTS. Any human, or subtype of human, will inherit this part hierarchy (unless the subtype has part hierarchies of its own which override this one). Each entity may be subdivided into parts in a number of different ways, and the part specialist supports any number of part hierarchies for each entity type. For example, humans can have another part hierarchy based on functionality of parts (so the top level parts would be *skeletal-system* , *digestive-system* , etc). Note: the part hierarchies are NOT used to generate all the parts of an individual, nor to determine the existence of the parts, although it may in some cases be used to determine the NON-existence of a part.

The next hierarchy (HUMAN-PARTS2) shows an alternative part breakdown of parts for a human. There may be any number of such part breakdowns, but all part hierarchies should have as their root an type which is an individual entity (e.g. *human* , *animal* , *chair*), not another part (e.g. *head* , *leg*). The same predicates may appear in different arrangements on the various part hierarchies, but each predicate may only appear once in any hierarchy.

```
HUMAN-PARTS2   Root: HUMAN   Hierarchy type: EXCLUSION    PART HIERARCHY
        PROPERTIES: (EXHAUSTIVE)    FUNCTION part hierarchy
HUMAN [1,29]   Exactly 1, for a total of exactly 1
|    HUMAN-OTHER [29,29]       A set of at least 0
|    NERVOUS-SYSTEM [24,28]    Exactly 1, for a total of exactly 1
|    |    NERVOUS-OTHER [28,28]        A set of at least 0
|    |    NERVES [27,27]       A set of at least 1, for a total of at least 1
```

```
|    |    SPINAL-CHORD [26,26]         Exactly 1, for a total of exactly 1
|    |    BRAIN [25,25]        Exactly 1, for a total of exactly 1
|    RESPIRATORY-SYSTEM [20,23]        Exactly 1, for a total of exactly 1
|    |    RESPIRATORY-OTHER [23,23]    A set of at least 0
|    |    TRACHEA [22,22]      Exactly 1, for a total of exactly 1
|    |    LUNG [21,21]         A set of exactly 2, for a total of exactly 2
|    REPRODUCTIVE-SYSTEM [19,19]       Exactly 1, for a total of exactly 1
|    CIRCULATORY-SYSTEM [14,18]        Exactly 1, for a total of exactly 1
|    |    CIRCULATORY-OTHER [18,18]    A set of at least 0
|    |    ARTERY [17,17]       A set of at least 1, for a total of at least 1
|    |    VEIN [16,16]         A set of at least 1, for a total of at least 1
|    |    HEART [15,15]        Exactly 1, for a total of exactly 1
|    SKELETAL-SYSTEM [8,13]    Exactly 1, for a total of exactly 1
|    |    SKELETAL-OTHER [13,13]       A set of at least 0
|    |    SKULL [12,12]        Exactly 1, for a total of exactly 1
|    |    LEG-BONES [11,11]    A set of at least 4, for a total of at least 4
|    |    ARM-BONES [10,10]    A set of at least 4, for a total of at least 4
|    |    SPINE [9,9]  Exactly 1, for a total of exactly 1
|    DIGESTIVE-SYSTEM [2,7]    Exactly 1, for a total of exactly 1
|    |    DIGESTIVE-OTHER [7,7]        A set of at least 0
|    |    TOOTH [6,6]  A set of at least 0 and at most 32, for a total between 0 and 32
|    |    SMALL-INTESTINE [5,5]        Exactly 1, for a total of exactly 1
|    |    LARGE-INTESTINE [4,4]        Exactly 1, for a total of exactly 1
|    |    STOMACH [3,3]        Exactly 1, for a total of exactly 1
```

Since parts are also types, it is possible for there to be subtypes of them, and therefore type hierarchies, which tangle with the part hierarchies. For example, the *finger* hierarchy below:

```
FINGER [1,6]
|    INDEX-FINGER [2,2]
|    THUMB [3,3]
|    RING-FINGER [4,4]
|    PINKY-FINGER [5,5]
|    MIDDLE-FINGER [6,6]
```

This allows assertions about specific types of fingers, and these will inherit all the part properties of *finger* .

The second part of the part specialist keeps track of part-of relationships which have actually been asserted. Eventually it may also keep track of relationships among the parts, like attachment. So far the part specialist can determine if a given part IS part of a particular entity, or another part if it has been asserted, or a set of transitive *part-of* inferences can be made that prove it (for example, given that *(hand1 part-of arm1)* and *(arm1 part-of body1)* , it can determine *(hand1 part-of body1)*). In addition, if the first part's type is a subpart of the second part's type, and the second part is unique (only one per entity), the part specialist can also answer that the first part is part of the second part (i.e. any *eye* is part of the *head*).

Parts which are on different entities, or different entity types cannot be part of each other. For example, a *leg* belonging to Little Red Riding Hood cannot also belong to the wolf, or to Sue. Also, parts whose types are disjoint cannot be part of each other, so a *finger* cannot be part of a *leg* (for a human).

There is a list of familiar parts ***familiar-parts*** which can be used to tell when a part CANNOT belong to an entity. A "normal level" is maintained for each of these parts, and when attempting to see if an entity can have such a part, if it does not exist on an exhaustive hierarchy which applies to

that entity by at least the "normal level", it is assumed that the part is not there. So for example, we could determine that Little Red Riding Hood does not have a tail, because in a FORM hierarchy the tail occurs at level 1, and after searching 1 level deep in the FORM hierarchies applicable to Little Red Riding Hood, not tail was found. The normal levels are set up automatically using the part hierarchies actually loaded. This occurs during the hierarchy numbering, so if you want to add to ***familiar-parts*** , you should do so before any numbering takes place (i.e. immediately before or immediately after the **add-part-hier** commands).

As for the type and predicate hierarchy specialists, the hierarchical information stored by the part specialist is applied in modally embedded contexts (both subnets and belief simulations) as well as at the top level (see page 95). However, the specialist does maintains subnets for part-of relationships. The part specialist has not yet been updated to accomodate multiple-environment reasoning, so information on part-of relationships between individuals should not be entered into the shared knowledge environment (there will be no ill effects, but the information will not be available for reasoning).

5.6 Episode Specialist

This specialist is used only to determine the relationships between **, * and @,and to return these relationships when requested. This makes the system a little more consistent, although special tests for the compatibility (or incompatibility) of the arguments still have to be done for propositions involving these predicates.

5.6.1 Using the Episode Specialist

The domain of the episode specialist consists of literals which contain the episodic predicates **, * and @. This specialist is only used to determine the relationship between those predicates.

This specialist (**'episode-specialist**) is automatically started up when **EPILOG** is loaded, so is always active.

Trace values for the episode specialist are:

bf episode-test Shows comparisons between episodic predicates

5.7 Time Specialist

The temporal specialist is used to accelerate question answering for questions about temporal relations between episodes (or events) and/or time points. When a proposition is asserted or questioned in **EPILOG** which is in the domain of the temporal specialist, the specialist is invoked to store or evaluate the proposition. For example,

(e1_episode before e2_episode)

would cause the temporal specialist to be invoked to save that relationship for future use.

The temporal specialist maintains its own representation of episodes and times, and uses this representation to achieve efficient temporal reasoning. Note that currently episodes are considered to be their time intervals for this implementation.

5.7.1 Using the Time Specialist

To start up the time specialist, issue the following command:

(use-spec 'time-specialist)

The sorts associated with temporal entities are:

episode - used for episodes, events, or intervals - indicating a possible span of time

time - used for "instants" of time (non-decomposable) or absolute times

The domain of the temporal specialist consists of literals with the following patterns:

Argument1	Predicate	Argument2	Argument3
Sort	Sort	Sort	
episode	**equal**	*episode*	
time	**same-time**	*time*	
episode	**before**	*episode*	*[episode]*
time	**after**	*time*	
	during		
	contains		
	overlaps		
	overlapped-by		
episode	**between**	*episode*	episode
time	time	time	
episode	**at-most-before**	*episode*	*number*
time	**at-least-before**	*time*	
	exactly-before		
	exactly-after		
	at-most-after		
	at-least-after		
episode	**has-duration**	*number*	

In all patterns except the second last group (with at-most...), a *time* argument may be either a named time point or an absolute time specification. For the exception, a time argument must be a named time point. *[]* indicates an optional argument of the given sort. Note that predicate stems only are given in the above table - each group includes all possible predicates with that stem. For example, the second pattern also includes **before-1, after-0, contains-0-1,** etc. The predicates will be described in more detail shortly.

The temporal specialist maintains a graph representation of time points. Events, episodes, and intervals are considered to have a start point and an end point, and all reasoning is done with those points, rather than the event/episode/interval itself. For convenience, the start and end points of an event e1 will be referred to as e1.start and e1.end in this document (in the time graph *start* is appended to the episode name to get the start point, and *end* for the end point).

Arcs between points indicate relationships, which may be strict or nonstrict. Strictness values on the predicates indicate whether a relationship is to be strict on nonstrict. Predicates look like *stem[-strict1[-strict2]]* .

Strictness	Meaning
0	there is no duration between (meets)
1	there is non-zero duration between (strict)
nil	either 0 or 1 (nonstrict)

If two points are equal, they are collapsed into one, so there are no arcs for "0" strictness.

In addition, points may have absolute times (minimum and maximum) associated with them. Arcs may have durations (minimum and maximum) associated with them. Absolute times and strictness values are propagated around the time graph to ensure that the information is as complete as possible.

5.7.1.1 Time Specialist Predicates

Following is a list of predicates that the time specialist knows about. Note that some of these predicates (equal) are not necessarily temporal in nature - the arguments used tell the specialist that they are.

Predicate	**Meaning (e1_episode predicate e2_episode)**
equal	$e1.start = e2.start$ and $e1.end = e2.end$
same-time	same as equal
before-0	$e1.end = e2.start$ (meets)
before-1	$e1.end < e2.start$ (strictly before)
before	$e1.end <= e2.start$
after-0	$e2.end = e1.start$ (met-by)
after-1	$e2.end < e1.start$ (strictly after)
after	$e2.end <= e1.start$
during-0-0	same as equal
during-1-0	$e1.start > e2.start$ and $e1.end = e2.end$ (ends)
during–0	$e1.start >= e2.start$ and $e1.end = e2.end$ (ends)
during-0-1	$e1.start = e2.start$ and $e1.end < e2.end$ (starts)
during-0	$e1.start = e2.start$ and $e1.end <= e2.end$ (starts)
during-1-1	$e1.start > e2.start$ and $e1.end < e2.end$
during-1	$e1.start > e2.start$ and $e1.end <= e2.end$
during–1	$e1.start >= e2.start$ and $e1.end < e2.end$
during	$e1.start >= e2.start$ and $e1.end <= e2.end$
contains-0-0	same as equal
contains-1-0	$e1.start < e2.start$ and $e1.end = e2.end$ (ended-by)
contains–0	$e1.start <= e2.start$ and $e1.end = e2.end$ (ended-by)
contains-0-1	$e1.start = e2.start$ and $e1.end > e2.end$ (started-by)
contains-0	$e1.start = e2.start$ and $e1.end >= e2.end$ (started-by)
contains-1-1	$e1.start < e2.start$ and $e1.end > e2.end$
contains-1	$e1.start < e2.start$ and $e1.end >= e2.end$
contains–1	$e1.start <= e2.start$ and $e1.end > e2.end$
contains	$e1.start <= e2.start$ and $e1.end >= e2.end$
overlaps-0-0	same as equal
overlaps-1-0	same as contains-1-0
overlaps–0	same as contains–0
overlaps-0-1	same as during-0-1
overlaps-0	same as during-0
overlaps-1-1	$e1.start < e2.start$ and $e1.end < e2.end$
overlaps-1	$e1.start < e2.start$ and $e1.end <= e2.end$
overlaps–1	$e1.start <= e2.start$ and $e1.end < e2.end$
overlaps	$e1.start <= e2.start$ and $e1.end <= e2.end$
overlapped-by-0-0	same as equal
overlapped-by-1-0	same as during-1-0
overlapped-by–0	same as during–0
overlapped-by-0-1	same as contains-0-1
overlapped-by-0	same as contains-0
overlapped-by-1-1	$e1.start > e2.start$ and $e1.end > e2.end$
overlapped-by-1	$e1.start > e2.start$ and $e1.end >= e2.end$
overlapped-by–1	$e1.start >= e2.start$ and $e1.end > e2.end$
overlapped-by	$e1.start >= e2.start$ and $e1.end >= e2.end$

Predicate	Meaning (e1_episode pred e2_episode e3_episode)
between-0-0	$e1.end = e2.start$ and $e2.end = e3.start$
between-1-0	$e1.end < e2.start$ and $e2.end = e3.start$
between–0	$e1.end <= e2.start$ and $e2.end = e3.start$
between-0-1	$e1.end = e2.start$ and $e2.end < e3.start$
between-0	$e1.end = e2.start$ and $e2.end <= e3.start$
between-1-1	$e1.end < e2.start$ and $e2.end < e3.start$
between-1	$e1.end < e2.start$ and $e2.end <= e3.start$
between–1	$e1.end <= e2.start$ and $e2.end < e3.start$
between	$e1.end <= e2.start$ and $e2.end <= e3.start$

Predicate	Meaning (e1_episode pred e2_episode d)
at-most-before	$e1$ $before$ $e2$ and maximum duration between is **d**
at-least-before	$e1$ $before - 1$ $e2$ and minimum duration is **d**
exactly-before	$e1$ $before - 1$ $e2$ and both maximum and minimum durations are **d**
at-most-after	$e1$ $after$ $e2$ and maximum duration between is **d**
at-least-after	$e1$ $after - 1$ $e2$ and minimum duration is **d**
exactly-after	$e1$ $after - 1$ $e2$ and both maximum and minimum durations are **d**

Predicate	Meaning (e1_episode pred d)
has-duration	$e1START$ $before$ $e1END$ and duration between is **d**

For all predicates except the last two groups (involving durations), some of the arguments may be absolute times. In that case, the maximum or minimum absolute time of the point involved is set or checked, rather than a relationship between points. For example

(e1_episode before (date 1987 04 01 00 00 00))

will set the absolute time maximum of e1.end to be the given time.

The first group of predicates (except **equal** and **same-time**) has an optional third argument which is the "timeframe" This argument must be an episode, and means that the other arguments are both during that episode (i.e. within that timeframe).

5.7.1.2 Time Specialist Functions

In addition to predicates, some functions are recognized by the temporal specialist. Arguments to these functions must all be constants, or functions with constant arguments. Each function returns a concept which is the result of the function. This can then be used as the argument to a predicate or another function.

Functions are evaluated with respect to the current subnet, which is the subnet in effect at the level the function is being evaluated at. For example, (john believes ((start-of e1) before (end-of e2))) will have the subnet for john used to calculate the start and end points.

The functions are:

(start-of *episode*) [*function*]

This function returns a concept which is the start of *episode* in the current subnet. If *episode* is not an episode (i.e. it is a time point), the point is returned (start of a point is the point itself).

(end-of *episode*) [*function*]

This function returns a concept which is the end of *episode* in the current subnet. If *episode* is not an episode (i.e. it is a time point), the point is returned (end of a point is the point itself).

(date *year month day hour minute second*) [*function*]

This function returns a quoted expression of the form

('time year month day hour minute second)

which is recognized by the temporal specialist as an absolute time.

(relation *arg1 arg2*) [*function*]

This function returns the most strict relation it can find between *arg1* and *arg2* in the current subnet. *Arg1* and *arg2* may be episodes, timepoints, or absolute times.

(elapsed *arg1 arg2*) [*function*]

This function returns the best duration bounds it can find between *arg1* and *arg2* in the current subnet. *Arg1* and *arg2* may be episodes or timepoints. The duration returned is a quoted expression of the form

(minimum maximum)

Note - this used to be called *duration* .

(duration-of *e1*) [*function*]

This function returns the duration between the start of an event and its end. It may also be used on assertion to set the duration bounds of an event. For example,

((duration-of e1_episode) less-than 4000)

5.7.1.3 Time Specialist Display and Controls

The following are display functions that can be called through the display command (all arguments are optional):

(display 'time-info *point subnet* **)**

Displays information for the given time point, or for all time points if no *point* given. The node name, its pseudo-time and absolute times are printed. If the *'-full* parameter is specified, the lists of ancestors and descendants (cross-chain and in-chain) are printed for each point.

(display 'meta-info *chain subnet* **)**

Displays information for the given chain, or for all chains if no *chain* given. If the *'-full* parameter is specified, the cross-chain link information is also printed; otherwise only the chain numbers, and first nodes on the chain are printed.

(display 'episode-info *event subnet* **)**

Displays information for the given episode, or for all episodes if no *episode* given if *'-brief* is used,

the episode start and end are listed only, if *'-full* , the time point info for them is also displayed.

For each of the above, the subnet is the string indicating which subnet - for example, / indicates *main* , **/john** is the subnet for *john* , **/john/mary** for the subnet for *mary* within *john* , etc. If no subnet is specified, the main net / will be used.

Although the time specialist has no tweakable parameters of its own, the user can still control how hard it works by tweaking the specialist interface parameters. An effort level (for either ***specialist-eval-effort*** or ***specialist-entry-effort***) of 0 means "constant time only" checking, which means that pseudo-time and absolute time comparisons can be done, but no metagraph searches will be. Anything higher than 0 indicates that a full blown graph search may be done. On entry, the entry effort value is used to do an internal search to find the existing relationship between points so that they may be collapsed if necessary. For example, if we have *(a before b)* in the timegraph already and we now enter *(a after b)* , these are not necessarily inconsistent and so will not be detected on input, but for the time graph to remain consistent, it must collapse this into *(a equal b)* . If input is guaranteed to be consistent, then ***specialist-entry-effort*** can be set to 0 without problems.

In addition, the parts of the time specialist that interact with the theorem prover can be controlled by the ***spec-enter*** , ***spec-evaluate*** and ***spec-compare-lits*** flags. For minimal use of the time specialist, the ***spec-enter*** and ***spec-evaluate*** flags should be on.

Trace values for the time specialist are:

time-entry - Traces input of temporal relations

time-point-entry - Traces input of point relations

time-eval - Traces evaluation of temporal relations

time-function-eval - Traces evaluation of temporal functions

time-point-eval - Traces evaluation of point relations

abs-time-entry - Traces entry of dates

abs-time-eval - Traces evaluation of date relationships

time-duration-entry - Traces entry of durations

time-duration-eval - Traces evaluation of durations

time-search - Traces metagraph searching

time-test - Traces comparisons of literals

time-all - Traces all time operations, including *time-entry, time-eval,time-point-entry, time-point-eval, abs-time-entry, abs-time-eval, time-duration-entry, time-duration-eval, time-search* , and *time-test* .

time - Traces interesting time stuff, including *time-entry, time-eval, time-point-entry, time-point-eval, abs-time-entry, abs-time-eval, time-duration-entry, time-duration-eval, and time-test* .

time-min - Traces basic time stuff, including *time-entry, time-eval,* and *time-test* .

5.7.2 Details of the Time Specialist

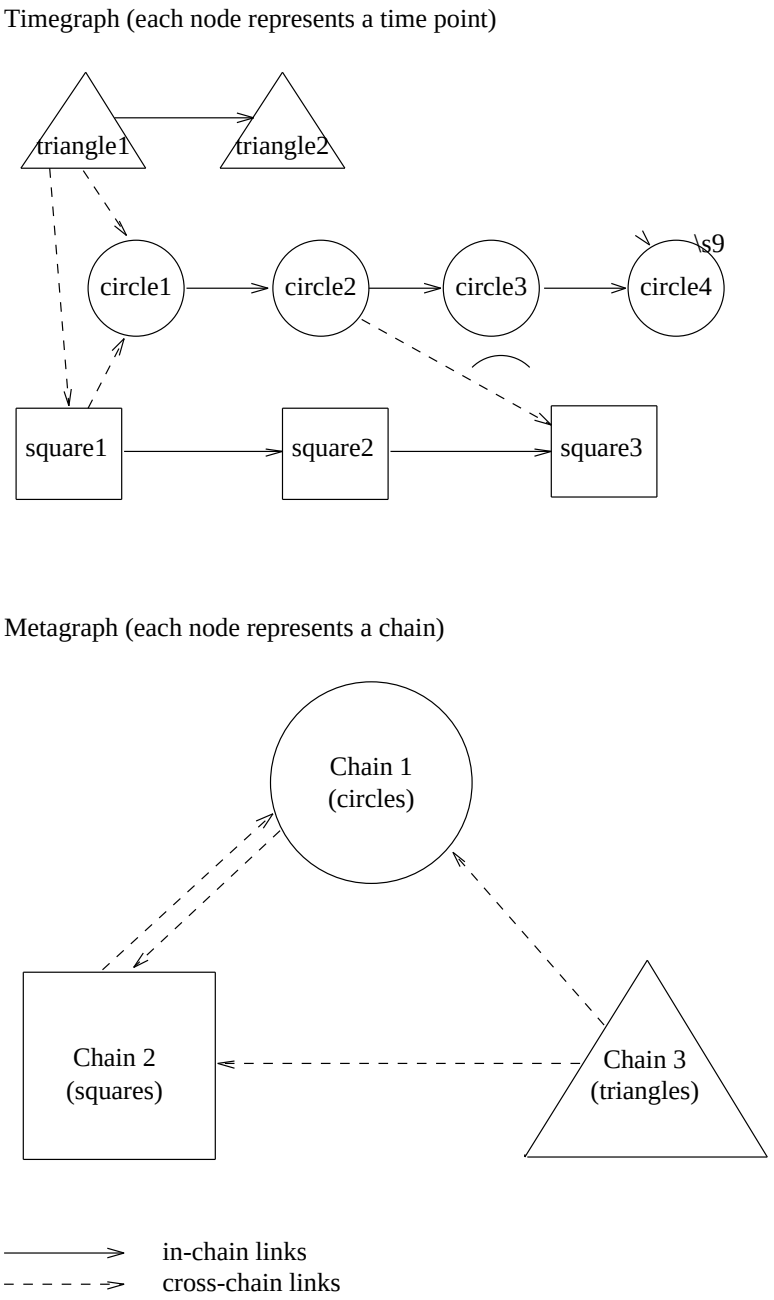
The time specialist implemented here is based on the specialist designed by J. Taugher [in his MSc thesis] and Len Schubert, with some enhancements to handle both strict and nonstrict relations, and combinations of events, time points and absolute times in propositions.

The representation used is a partial order graph that has been partitioned into "chains". All the points belonging to a chain are linearly ordered with respect to each other. There may be transitive arcs between the points in a chain. Cross chain links define relations between points in one chain and points in another.

For points within a chain, an arbitrary pseudo-time number is associated with each point. These numbers show the ordering relationship between points in a chains. In addition, the minimum point and the maximum point on the chain that a point can be equal to are stored with it - giving a range of points that can possibly be equal. These are used to show whether the relationship given by the psuedo times alone is strict or nonstrict (for example, < or <=). Determining the relationship between any two points in the same chain can be done in constant time using these pseudo-times, while a graph search is required if they are on different chains.

In addition to the time graph of time points, there is a metagraph of chains. The cross chain links define arcs between chains in the metagraph. The metagraph is used to search for paths from one point to another. This makes a graph search dependant on the number of cross chain links rather than the total number of time points (a significant savings).

Figure T1 shows an example time graph and meta graph.



T1 Example Timegraph and Metagraph

In the time graph, small circles represent points on chain 1, small squares are points on chain2, and small triangles represent points on chain 3. So circle1 is before circle2, circle2 before circle3 and so on. There are cross chain links from triangle1 to circle1 (i.e. triangle1 is before circle1), from triangle1 to square1, from square2 to circle1, and one more from circle2 to square3.

In the metagraph, these cross chain links show up as links between meta-nodes. There is one meta node for chain 1 (the big circle), one for chain 2 (the big square) and another for chain 3 (the big triangle).

The links within chains do not show up here, as within a chain they are not needed to determine relations. Following the cross chain links, we can get that triangle1 is before square3, and square1 is before circle3, but no information about triangle2 and square3.

Furthermore, an absolute time (date) minimum and maximum are stored with each time point. These are a six-tuples of the form *(year month day hour minute second)* , where each element may be numeric or symbolic (e.g. *(1987 04 a 12 b c)* represents some time at or after 12 a.m. and before 1 p.m. of some day in April, 1987). Symbolic information may be filled in later by another assertion, or left unspecified throughout the session. Absolute time maxima propagate back to points before the given point (in the chain or on other chains), and minima propagate forward. This ensures that each point has the best absolute time information possible. Absolute time comparisons can sometimes be used to get a relation in constant time between two points on different chains, avoiding a metagraph search.

When an absolute time is not completely specified, the time specialist will ask for either the *min-of* or the *max-of* the symbolic element, as appropriate. The specialist interface passes this evaluation request along to a specialist that can evaluate it (currently the number specialist can), and uses any result it gets back to more completely specify the absolute time. In addition, the symbolic element is noted as being important to the time specialist, so this literal and the time specialist are added to the interested party list for the concept.

Insertion time into the graph is constant in most cases, except for propagation of absolute times or strictness values, and for consistency checking if the entry effort level is greater than 0. In the worst case, propagation may require going to every point in the graph, although it is highly unlikely. Occasionally a chain may have to be renumbered, which requires going to all the points in a single chain.

Duration minima and maxima (in seconds) are stored on the links between points. These may affect the absolute times around them, which are then propagated. They are also used in calculating duration between points where the path uses this link. Durations may be unspecified and are then treated similar to unspecified abosolute times, generating an evaluation request and adding to the interested party list of the concept. To determine the duration betweeen any two points, an exhaustive search must be done between those points, calculating the duration along all paths to get the best one. This particular search uses a traditional depth first search over the entire time graph, rather than using the metagraph. Both duration information on arcs, and duration information implicit in absolute times are used.

Entry and evaluation of temporal literals only uses those literals with constant arguments. Comparisons between literals also allow variables.

To determine incompatibility (or compatibility) of literals, unification is done with every combination of arguments. After unification, variables are treated exactly the same as constants. One literal is then entered into the timegraph, and then the other is compared with the timegraph to see if it is consistent. If not, the two literals are incompatible. This is repeated for each possible unification. If all the arguments are constants, just one iteration is done, as there are no substitutions to be done. This aspect of the time specialist can be "turned off" by tweaking the parameter ***spec-compare-lits*** to nil.

When the belief specialist is active, and there are therefore multiple reasoning environments active simultaneously, time graph information stored in the current agent-specific environment "shadows," or overrides, information about the same nodes in the shared knowledge environment. When non-shared information is entered about a node that already exists in the shared knowledge environment, a copy of the node's entire is created in the non-shared environment, and then modified as necessary. Absolute time information is propagated only within a single environment, not from a node in one environment to a node in another. This is done to prevent non-shared information from leaking into the shared knowledge environment. See (Kaplan 2000) for more details on how the time specialist has been adapted

to multiple-environment reasoning.

5.8 Number Specialist

The number specialist is used to accelerate question answering for questions about relations between named number constants and numbers, and some arithmetic operations on them. When a proposition is asserted or questioned in **EPILOG** which is in the domain of the number specialist, the specialist is invoked. For example,

(n1_number lt= '5)

would cause the number specialist to be invoked to save that relationship for future use.

5.8.1 Using the Number Specialist

To start up the number specialist, issue the following command:

(use-spec 'number-specialist)

The sorts associated with number entities are:

real - used for any real number

integer - used for any integer

number - also used for integers

The domain of the number specialist consists of literals with the following patterns:

Argument1 Sort	**Predicate**	**Argument2 Sort**
real	**equal**	*real*
integer	**lt**	*integer*
	less-than	
	gt	
	greater-than	
	lt=	

The number specialist maintains a graph representation of named number points. Arcs between points indicate relationships, which may be strict (lt or $<$) or nonstrict ($lt =$ or $<=$). If two points are equal, they are collapsed into one, so there are no arcs for "0" strictness.

5.8.1.1 Number Specialist Predicates

Following is a list of predicates that the number specialist knows about. Note that some of these precicates (equal) are not necessarily only numeric in nature - the arguments used tell the specialist that they are.

Predicate	**Meaning (n1_number predicate n2_number)**
equal	$n1 = n2$
lt	$n1 < n2$
less-than	$n1 < n2$
gt	$n1 > n2$
greater-than	$n1 > n2$
lt=	$n1 \geq n2$
gt=	$n1 \leq n2$

5.8.1.2 Number Specialist Functions

In addition to predicates, some functions are recognized by the number specialist. Arguments to these functions must all be constants, or functions with constant arguments. Each function returns a concept or quoted expression which is the result of the function. This can then be used as the argument to a predicate or another function.

Functions are evaluated with respect to the current subnet, which is the subnet in effect at the level the function is being evaluated at. For example, (john believes ((max-of n1) < (min-of n2))) will have the subnet for john used to determine the minimums and maximums points.

The functions are:

(min-of *numbers*) [*function*]

This function returns the minimum of its arguments *numbers* . If any are symbolic names, the minimum on that node in the number graph is used in place of the name.

(max-of *numbers*) [*function*]

This function returns the maximum of its arguments *numbers* . If any are symbolic names, the maximum on that node in the number graph is used in place of the name.

(value-of *concept*) [*function*]

This function returns a quoted expression consisting of a single value, if known for *concept* , or two values in a quoted expression corresponding to the minimum and maximum of the number *(minimum maximum)* .

(relation *arg1 arg2*) [*function*]

This function returns the most strict relation it can find between *arg1* and *arg2* in the current subnet.

(add *numbers*) [*function*]

This function returns the result of adding together all the arguments. If any are symbolic names, the value associated with that number is obtained from the number graph and used. If no exact value is known, the function will return nil.

5.8.1.3 Number Specialist Display and Control

The following are display functions that can be called through the display command (all arguments are optional):

bexmp0.5in(display 'number-info *point subnet*) Displays information for the given number, or for all numbers if no *point* given. This includes the minimum and maximum values of the points. If *'-full* is specified, descendants and ancestors of points are also printed.

For the above function, the subnet is the string indicating which subnet - for example, / indicates *main* , **/john** is the subnet for *john* , **/john/mary** for the subnet for *mary* within *john* etc. If no subnet is specified, the main net / will be used.

Although the number specialist has no tweakable parameters of its own, the user can still control how hard it works by tweaking the specialist interface parameters. An effort level (for either ***specialist-eval-effort*** or ***specialist-entry-effort***) of 0 means "constant time only" checking, which means that comparisons between minimum and maximum values may be done, but no graph searches will be. Anything higher than 0 indicates that a full blown graph search may be done.

In addition, the parts of the number specialist that interact with the theorem prover can be controlled by the ***spec-enter*** , and ***spec-evaluate*** flags. For minimal use of the number specialist, these flags should be on.

Trace values for the number specialist are:

number-entry - Traces entry of number relations

number-prop - Traces propagation of number relations

number-eval - Traces evaluation of number relations

number-function-eval - Traces evaluation of number functions

number-search - Traces searching through the number graph

number-test - Traces comparison of literals using the number specialist

number-all - Traces all number operations, including *number-entry, number-eval, number-prop, number-test* , and *number-search* .

number-min - Traces basic number operations, including *number-entry, number-test,* and *number-eval* .

5.8.2 Details of the Number Specialist

The number specialist implemented here uses a simple graph representation to represent and reason with numeric relations. Nodes represent named number constants, and arcs between (which may be strict < or nonstrict <=) represent the relation between the points. The type of number (integer or real), a maximum, minimum and exact value (if known) are associated with each point. Maxima propagate backward, minima forward, and values are either set explicitly by the predicate **equal** or when the minimum and maximum are equal. When a named number is asserted less than (or less than or equal) to a number constant, the maximum of the number is set according to whether it is real or integer, and whether the constant is real or integer. For example,

(n1_integer lt 3.5)
sets the maximum of *n1* to 3, and

(n2_real lt= 4)
sets the maximum of *n2* to 4.0. Minima are handled similarly, by asserting that a number is greater than some constant.

The maximum for a real number (and similarly, the minimum) is only a single number, with no information about whether this "endpoint" is included in the range of possible values for the number. This can cause some confusion when testing at these boundaries - for example, asserting that a number is less than or equal to 3.2 will result in "yes" being answered when we ask if that number is less than 3.2 as well. In natural language understanding, which is what this system is intended for, this problem rarely occurs. Integers are handled exactly, as the "next lowest" or "next highest" number is always known, so the bounds mean the same for all of them.

To determine the relation between two points, or the validity of a given relation, the minima and maxima of the numbers are first compared, and then an exhaustive graph search is done if necessary. For the use of numbers in this system, this search will rarely have to be done, and if it is, only a few steps will be required.

When determining incompatibility of literals the same methods are used that are used in the time specialist, although there are fewer predicates to consider. As each literal has exactly two arguments, there are only two possible unifications that can be tried for the resolving and factoring attempts: 1st of literal1 with 1st of literal 2, 2nd with 2nd, and 1st of literal1 with 2nd of literal 2, 2nd of literal1 with 1st of literal2. For example, *(n1 lt n2) vs (x gt y)* could give resolving actions with unifications: x/n1, y/n2, and x/n2, y/n1.

The number specialist has not yet been updated to accomodate multiple-environment reasoning, so information on numerical relationships should not be entered into the shared knowledge environment (there will be no ill effects, but the information will not be available for reasoning).

5.9 Color Specialist

The domain of the color specialist consists of literals which contain color predicates. The specialist determines the relationships between simple color predicates, and those operated on by some predicate operators (operators which "hedge" the color - like *sort-of*).

5.9.1 Using the Color Specialist

To start up the color-specialist, issue the following command:

(use-spec 'color-specialist)

The current color predicates recognized are: **white, black, blue, red, yellow, green, purple, orange, brown, pink, grey, chartreuse, magenta, bluegreen, lead, salmon, beige, tan, crimson, aqua,** and **rust**

Of these, *white, black, blue, red, yellow, green, purple, orange, brown, pink,* and *grey* are "basic" colors. They completely partition the color space. These colors have the largest ranges. Any other colors have smaller ranges, usually near or overlapping a border between basic colors.

To add more predicates to the color specialist, do the following:

Determine the region boundaries by deciding on ranges for the purity, dilution and hue. To add a new color, choose numbers for purity, dilution and hue that keep the appropriate relative order between this color and neigboring colors. For example, to add **rust-red** , choose numbers that make the corresponding region overlap (and extend about 1/4 way into) the brown and red regions, not too light and not too dark. The color definitions are located in the file *color/colors* for comparison.

Use the following function to add the color:

(make-color *color purity-min purity-max dilution-min dilution-max hue-min hue-max***)**

This can be called interactively as **color:make-color** (if the color specialist is already active). Alternatively, the call to **make-color** can be added to the file *color/colors* , and the color will be added whenever the color specialist is invoked. No package names are needed on the color for either call.

Note that the operator *sort-of* may act on a color predicate. The color specialist handles this as well. A hedged color is assumed to have range boundaries twice their normal distance.

Note: there are some cases where the color specialist gets different answers depending on the order it compared the predicates in, and sometimes it gets inferences that don't seem "natural". Please note any occurrences of these and report them to the authors.

5.9.1.1 Color Specialist Display and Controls

The following display function can be called through the display command (all arguments are optional):

(display 'color-info *color* **)**

Displays the ranges of purity, dilution and hue for *color* , if a color is given. Otherwise, if the flag *-brief* is used, a list of the known color names will be displayed. If *-full* is used, the ranges of purity, dilution and hue for all the known colors will be displayed.

The color specialist has two tweakable parameters:

color-margin which determines how different two range boundaries can be to be considered equal. This is initially set to 0.01, but can be changed using the **tweak** command.

color-hedged-operators which indicates the operators on predicates that make them into "hedged" predicates. It is initially set to the single operator *sort-of* .

The other control available on this specialist is ***spec-compare-preds*** , which should not be turned off except for specific testing cases.

Trace values for the color specialist are:

color-test - Traces comparison of predicates using the color specialist

color-details - Traces details of comparison of predicates using the color specialist

5.9.2 Details of the Color Specialist

The color specialist determines the relationships between color predicates. It uses a cylindrical color model developed by Mary Angela Papalaskaris. This representation was arrived at by imagining that any color is composed of some amount of a pure, monochromatic color, plus certain amounts of black and white.

There are three dimensions to this object (from *Accelerating Deductive Inference: Special Methods for Taxonomies, Colours and Times* , by Schubert, Papalaskaris and Taugher):

hue - this dimension runs through the continuum of ranbow hues, arranged in a circle and arbitrarily scaled from 0 to 12

purity - the radial axis - parametrizes the amount of black present

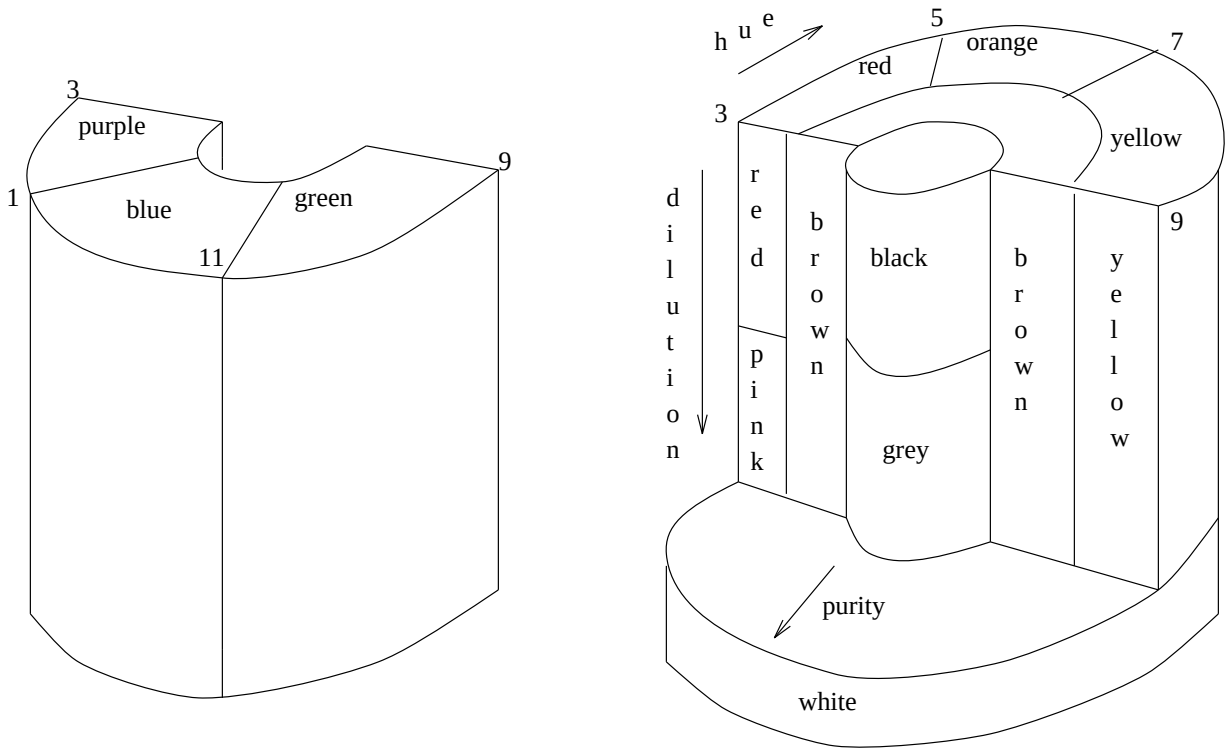
purity = pure color / (pure color + black)

which decreases from 1 to 0 as black is added

dilution - axial dimension - parametrizes the amount of white present

dilution = white / (pure color + black + white)

which increases from 0 to 1 as white is added.



Color cylinder with the "cool" shades lifted away.
Adapted from *Accelerating Deductive Inference ...* , p 42.

Both simple color predicates, and hedged color predicates (predicates which have an operator like *sort-of* acting on them) can be compared by the specialist.

Note: there are some cases where the color specialist gets different answers depending on the order it compared the predicates in, and sometimes it gets inferences that don't seem "natural". Please note any occurrences of these and report them to the authors.

5.10 Equality Specialist

The equality specialist is used to both to accelerate question answering for questions about equality, and to maintain equivalence sets for use internally in EPILOG (which further accelerates things). The specialist maintains equivalence sets of equal constants, which it transmits back to EPILOG, as well as sets of "non-equal" items. When an equality, or inequality, is detected, the equality specialist is invoked. Note that this specialist also accepts negative information (unlike most of the others). For example,

(c1 equal chair1)

would cause the equality specialist to be invoked to save that relationship for future use by both the equality specialist and the EPILOG core.

(not c1 equal chair5)

would also be saved in the equality specialist.

The equality specialist is automatically started up when EPILOG starts up.

The domain of the equality specialist consists of literals with the following pattern:

Argument1 Sort	**Predicate**	**Argument2 Sort**
	equal	

In the above patterns, **equal** takes two arguments of any sort.

5.10.1 Equality Specialist Functions

A few functions are included, just so that equivalence information can be extracted by a function, if desired.

Functions are evaluated with respect to the current subnet, which is the subnet in effect at the level the function is being evaluated at. For example, (john believes (h1 equal h2)) would have the subnet for john used to determine the contents and therefore the union of the two sets, friends and enemies.

The functions are:

(equivalence-members *item*) [*function*]

This function returns a record which is the set of known items equivalent to *item* (including *item*).

(non-equal-members *item*) [*function*]

This function returns a record which is the set of known items not equal to *item* .

5.10.2 Equality Specialist Display and Controls

The following are display functions that can be called through the display command (all arguments are optional):

(display 'equality-info *item subnet* **)**
Displays equivalence and non-equal information for the given item, or lists all known sets of equivalent items if no *item* is given. The subnet is the string indicating which subnet to display from - for example, / indicates *main* , **/john-believe** for John's beliefs, etc) If no subnet is specified, the main net / will be used. **Equality-info** can be abbreviated to **equal-info** .

The parts of the equality specialist that interact with the theorem prover can be controlled by the ***spec-enter*** , ***spec-evaluate*** and ***spec-compare-lits*** flags. For minimal use of the equality specialist, the ***spec-enter*** and ***spec-evaluate*** flags should be on.

The specialist has the option of assuming that all user given names (i.e. non-skolem constants) are unique. This flag is called ***unique-names-assumptions*** and defaults to *t* . If *t* , two individual names will be assumed to be not equal. If set to *nil* , they could be equal, and a question asking this would be answered *unknown* .

Trace values for the equality specialist are:

equality-entry - Traces entry of equivalence and non-equal relationships

equality-eval - Traces evaluation of equivalence and non-equal relationships

equality-function-eval - Traces evaluation of equality functions

equality-test - Traces comparison of literals within the equality specialist.

equality-min - Traces interesting things in the equality specialist, including *equality-eval, equality-entry, equality-function-eval* and *equality-test* .

equality-all - Traces everything about the equality specialist, including *equality-eval, equality-entry, equality-function-eval* and *equality-test* .

5.10.3 Details of the Equality Specialist

The equality specialist uses the same low level set structures used by the set specialist. Positive equalities are saved by maintaining equivalence sets, and transmitting these back to a special table in EPILOG where it may accesss the information. Negative equalities are saved by maintaining a set of items not equal to a given item. These are not transmitted to EPILOG.

When testing equality, the two arguments are first compared to see if they are identical. If not, the equivalence sets are checked to see if one is a member of the other's equivalence set. Then non-equal sets are checked to see if the evaluation should terminate with NO. If none of these gives a result, the sorts of the objects are checked to see if they are incompatible. As a last resort, the ***unique-names-assmuption*** can be used to say that two non-equal user-named constants are not equal.

5.11 Set Specialist

The set specialist is used to accelerate question answering for questions about set membership. The sets handled by this specialist are not sets according to the mathematical definition - they are more like collections. When a proposition is asserted or questioned in et which is in the domain of the set specialist, the specialist is activated. For example,

(apple1 member-of fruits_set)

would cause the set specialist to be invoked to save that relationship for future use.

5.11.1 Using the Set Specialist

To start up the set specialist, issue the following command:

(use-spec 'set-specialist)

The sorts associated with set entities are:

set - used for a set

The domain of the set specialist consists of literals with the following pattern:

Argument1 Sort	**Predicate**	**Argument2 Sort**
set	**equal**	*set*
	member-of	*set*
	member-of-0	
set	**subset-of**	*set*
set	**has-cardinality**	*integer*
set	**(coll** *predicate)*	
	((coll-of *number* **)** *predicate)*	

In the above patterns, **equal** takes two arguments sort set (actually, only 1 is required to be of sort set), while **member-of** and **member-of-0** can take any argument sort for the first argument, but the second must be a set. **Subset-of** takes two arguments, both of which must be of sort *set.* Note that *member-of-0* is accepted only for evaluation, as it is inherently ambiguous and requires assumptions to enter. **Has-cardinality** takes a *set* argument and an *integer* argument. Cardinality may also be asserted using the function **cardinality-of** (see below). The operators *coll* and *coll-of* trigger this specialist regardless of the predicate operated on. The set specialist keeps track of named sets, their contents, cardinality and member types. In the above list of predicates, **member-of** is used for the top level contents of a set, whereas **member-of-0** refers to the recursion involved in nested sets. For example, if set *s1* consists of members *(apple1, orange1)* and set *s2* has members *(s1 grape1)* , then

(q '(apple1 member-of s2))

could not be answered affirmatively, while

(q '(apple1 member-of-0 s2))

would be.

The predicates will be described in more detail shortly.

Sets may be completely known (all members are known, and represented in the set specialists area), or partially known (there may or may not be other members of the set which we don't know about).

When setting up a set's contents using a proposition like *(s1 equal (set-of member1 member2))* , the set is considered fully known, whereas setting contents using *((set-of member1 member2) subset-of s1)* makes the set partially known. When testing to see if something a a member of a set, if it is not, and the set is fully known, the specialist can answer **no** , whereas if the set is only partially known, the specialist must answer **unknown** as that entity might be as as yet unknown member of the set.

When a function acts on a set, or a incompatibility/compatibility test is attempted, only the currently known contents of the set are used. For example, if a union function is done on two sets, the resulting union is independent of the two sets - any further additions of members to them will not be reflected in the union.

5.11.1.1 Set Specialist Predicates and Operators

Following is a list of predicates and operators that the set specialist knows about.

Predicate	**Meaning (a1 predicate a2)**
equal	*a1* = *a2*
member-of	*a1* is a direct member of set *a2* (top level)
member-of-0	*a1* is an indirect member of set *a2* (some level below)
subset-of	set *a1* 's direct members are a subset of set *a2* 's direct members (top level)
has-cardinality	*a1* has exactly *a2* top-level members.
(coll *predicate* **)**	*a1* is a set whose members are all predicate (no *a2*)
((coll-of *number* **)** *predicate* **)**	*a1* is a set whose members are all predicate, and whose cardinality is *number*

For the last two entries in the list, in addition to saving the information, the specialist makes the assertion *(A x (x member-of a1) (x predicate))* .

5.11.1.2 Set Specialist Functions and Term-Forming Operators

In addition to predicates, some functions and term-forming operators are recognized by the set specialist. Arguments to the functions must all be constants, or functions with constant arguments. Arguments to the term-forming operators are somewhat less restrictive, but if variables are involved they will not be evaluated. Each function returns a concept or quoted expression which is the result of the function. This can then be used as the argument to a predicate or another function.

Functions are evaluated with respect to the current subnet, which is the subnet in effect at the level the function is being evaluated at. For example, (john believes ((union-of friends enemies) equal ...)) have the subnet for john used to determine the contents and therefore the union of the two sets, friends and enemies.

The functions are:

(union-of-members *sets*) [*function*]

This function returns the union of all current members of its arguments *sets* If any are symbolic names, the contents of that set is obtained.

(union-of-members-0 *sets*) [*function*]

This function is identical to **union-of** except that the set contents are obtained recursively to give all the lowest level members.

(intersect-members *sets*) [*function*]

This function returns the intersection of the current members of its arguments *sets* If any are symbolic names, the contents of that set is obtained.

(intersect-members-0 *sets*) [*function*]

This function is identical to **intersect** except that the set contents are obtained recursively to give all the lowest level members.

(set-diff-members *set1 set2*) [*function*]

This function returns the set difference between the current top level members of *set1* and the current top level members of *set2* .

(set-diff-members-0 *set1 set2*) [*function*]

This function is identical to **set-diff** except that the set difference is taken between the current indirect members of the sets by recursively retrieving all members of nexted sets.

(set-of *members*) [*function*]

This function builds a set with contents *members* and returns a quoted expression of the form *'('set members)* which is recognized by the set specialist as a set.

(set-of-all *predicate*) [*function*]

This term-forming operator returns the unique name for a set whose members are all of the entities of type *predicate* . An assertion is made of the form *(A x ((x member-of name) <=> (x predicate)))* .

(union-of *sets*) [*function*]

This function returns the unique name for the set which is the union of *sets* . This linkage is recorded in the set connection records and used during evaluation, and to "propagate" information where required.

(intersect *set1 set2*) [*function*]

This function returns the unique name for the set which is the intersection of *set1* and *set2* . This linkage is recorded in the set connection records and used during evaluation, and to "propagate" information where required.

(set-diff *set1 set2*) [*function*]

This function returns the unique name for the set which is the difference between *set1* and *set2* .

This linkage is recorded in the set connection records and used during evaluation, and to "propagate" information where required.

(number-members *set*) [*function*]

This function returns the number of known direct members *set* contains.

(number-members-0 *set*) [*function*]

This function returns the number of known indirect members *set* contains by recursively counting all members of nested sets within *set* .

(set-members *set*) [*function*]

This function returns the current known direct members of *set* in the form of a quoted expression *'('set member1 member2 ..)* .

(set-members-0 *set*) [*function*]

This function returns the current indirect members of *set* in the form of a quoted expression *'('set member1 member2 ..)* by recursively getting all members of nested sets in *set*

(cardinality-of *set*) [*function*]

This function returns the cardinality of the given set. It can be used in assertion to set the cardinality - for example

((cardinality-of s1_set) less-than 20)

5.11.1.3 Set Specialist Display and Controls

The following are display functions that can be called through the display command (all arguments are optional):

(display 'set-info *setname subnet* **)** Displays information for the given set, or for all sets if no *setname* is given. The subnet is the string indicating which subnet to display from - for example, / indicates *main* , **/john** is the subnet for *john* , **/john/mary** for the subnet for *mary* within *john* , etc. If no subnet is specified, the main net / will be used.

The parts of the set specialist that interact with the theorem prover can be controlled by the bf *spec-enter* , ***spec-evaluate*** and ***spec-compare-lits*** flags. For minimal use of the set specialist, the ***spec-enter*** and ***spec-evaluate*** flags should be on.

The set specialist makes assertions about members and their types to EPILOG. This can be turned off by tweaking the ***set-assert*** flag to nil.

Trace values for the set specialist are:

set-entry - Traces entry of set membership and set equality relationships, and cardinality information.

set-eval - Traces evaluation of set membership and set equality relationships,and cardinality questions.

set-function-eval - Traces evaluation of set functions

set-test - Traces comparison of literals within the set specialist.

set-assert - Traces assertions made by the set specialist.

set-min - Traces interesting things in the set specialist, including *set-eval, set-entry, set-assert* and *set-test* .

set-all - Traces everything about the set specialist, including *set-eval, set-entry, set-assert* and *set-test* .

5.11.2 Details of the Set Specialist

The set specialist handles assertions and simple questions about set membership and set equality. The sets handled by this specialist are not sets according to the mathematical definition - they are more like collections. Membership relations, cardinality information, and equalities among sets are saved in the specialist's own representation for future use.

The set specialist implemented here keeps track of named sets, their contents, member type, and their cardinality in a hash table. Equalities between sets are saved by making both entries of the hash table refer to the same set description. Equalities between sets are true if both sets refer to the same set description, or if they have identical members, member types, and the same number of members.

When determining membership, union of members, intersection of members, set members, set difference of members, and so on, a -0 ending indicates that the members should be determined recursively by expanding any element that is itself a set. Only the lowest level elements (bottom of the tree of nested sets) are used. Without this flag, only the top level - i.e. the actual contents associated with the set (so the set name, instead of its contents) are used. When determining whether or not an item is a member of a set, or a set is a subset of another set, the set member types are considered as well.

A flag is associated with each named set indicating whether or not it is fully known. When using the contents of these sets, a + is appended to the list of members for partially known sets so that late stages know that the set is not complete. This is for internal use only, but does show up in some trace information.

Set intersection, union, set difference, and subset-of relations are handled by connecting the related sets together with a link. When new information is added to one of the sets, it can be propagated (if appropriate) to the other sets. For example, if *s1* is the intersection of *s2* and *s3* , if a member is added to *s1* , it must also be a member of both *s2* and *s3* (by definition of intersection), so those sets are also updated.

When determining incompatibility/compatibility of literals, if both predicates are membership predicates (**member-of** or **member-of-0**), or both are subset predicates (**subset-of**), set-set literal comparison is done. For membership, we try to unify the first argument of each literal. If they unify, the second argument of each (which is a set) is combined in the appropriate way to give a set residue. For example, when determining incompatibility of

(x member-of (set-of apple banana orange))

against

(y member-of (set-of apple orange grape))

x and y will unify (substituting y for x) . Then the intersection of those two sets is what we want for the residue, so we would get *(y member-of (quote (set apple orange)))* as the residue. For compatibility testing, the union of the two sets is used. If one is negated, we use the set difference of the two; if both are negated, incompatibility uses the union, compatibility the intersection, and the residue is negated.

For subset tests, we can either unify the first two arguments and do a similar operation to the membership tests, or we can unify the second two arguments, and then calculate the union of the sets specified by the first arguments (unless there is a negation, and then we use set-difference or intersection).

There are still some unresolved problems in literal comparison with the membership and subset predicates (there are more possibilities than this specialist actually handles).

5.12 String Specialist

The string specialist is used to accelerate question answering and verification during forward inference for literals about relations between strings, and functions involving strings. When a proposition is questioned or a formula involving a function is asserted in **EPILOG** which is in the domain of the string specialist, the specialist is invoked.

5.12.1 Using the String Specialist

To start up the string specialist, issue the following command:

(use-spec 'string-specialist)

The sorts associated with string entities are:

string - used for any string

The domain of the string specialist consists of literals with the following patterns:

Argument1 Sort	**Predicate**	**Argument2 Sort**
string	**string=**	*string*
	string-equal	
	string<	
	string>	
	string<=	
	string>=	
	string/=	
	string-lessp	
	string-greaterp	
	string-not-lessp	
	string-not-greaterp	
	string-not-equal	
	string-contain	

5.13 3 "String Specialist Predicates"

The "predicates" *string=, string-equal, string/=, string-not-equal, string<, string<=, string-lessp, string-not-lessp, string>, string>=, string-greaterp,* and *string-not-greaterp* are all Lisp functions. The string specialist just calls the appropriate Lisp routine and interprets the answer as a YES or NO.

String-contain is not a Lisp function however. It returns YES if the second argument is contained in the first argument. It is easy to add new "predicates" to this specialist - just make sure to modify both *string/definition.lisp* and add new code in *string/eval.lisp* .

5.13.0.1 String Specialist Functions

In addition to predicates, some functions are recognized by the string specialist. Arguments to these functions must all be constants, or functions with constant arguments. Each function returns a string result. This can then be used as the argument to a predicate or another function.

Many of the functions are just the Lisp string functions. These work exactly as described in the Lisp manual, but all the arguments must be strings - lists or quoted structures cannot be handled now (even though some of the Lisp functions themselves can handle them). The following Lisp functions are recognized: *string-trim, string-left-trim, string-right-trim, string-upcase, string-downcase,* and *string-capitalize* . In addition, there are some new functions. They are:

(string-field *string integer*) [*function*]

This function returns the portion of string which can be considered its first field. The default field divider character is "-", but this may be changed or added to by tweaking ***string-divider*** .

(sub-string *string start end*) [*function*]

This function returns the substring of *string* consisting of the elements in *string* from position *start* to position *end* , inclusive.

(string-concat *string1 string2 ...*) [*function*]

This function returns a string consisting of all the strings in its argument list concatenated together. Optionally, a character may be added between each string, by tweaking ***string-separator*** .

(string-number *string*) [*function*]

This function returns the number corresponding to *string* if string is composed entirely of digits, nil otherwise. (e.g. (string-number "321") returns 321).

It is easy to add new functions to the specialist - just update the file *string/definition.lisp* and add the new function to the file *string/functions.lisp* , patterning it after one of the existing functions there. (Note: any new functions or predicates should also be sent to the authors of the manual to incorporate into the string specialist so that you don't have to make the changes all over again when the next version of the system comes.)

5.13.0.2 String Specialist Display and Control

Because the string specialist does not store any information, there is nothing to display in it.

It does however have two tweakable parameters:

string-separator
: May be nil, or a single character. If a single character, this character is inserted between every pair of strings when they are concatenated using the function *string-concat* . This is initially set to nil, so any strings concatenated together will have no intervening blanks or other characters.

string-dividor
: This is a list of characters which may be used to separate fields for the *string-field* command. Initially it is set so that only the character "-" can separate fields (i.e. set to '(#–)), but it can easily be changed to include spaces or other special characters as well.

In addition, the parts of the string specialist that interact with the theorem prover can be controlled by the ***spec-evaluate*** flag. For minimal use of the string specialist, this flag should be on.

Trace values for the string specialist are:

string-eval - Traces evaluation of literals involving string relation

string-function-eval - Traces evaluation of functions involving strings predicates.

5.13.1 Details of the String Specialist

The string specialist is just a simple interface to the Lisp string functions, and a few new string functions. The system has the capability to handle strings as arguments in formulas, and this specialist allows the user to manipulate and test these strings. The specialist can only be used for literal and function evaluation now - it does not store any specific discourse information. For example,

("Hello there" string-contain "there")

would be evaluated by the string specialist to YES, and

(item1 has-field-1 (string-field "abc-def" 1))

would be simplified to

(item1 has-field-1 "abc")

5.14 Belief Specialist

The belief specialist stores and evaluates literals of the form **(*t1* believe *t2*)**, where *t1* is a ground term and *t2* is either a **that**-term, *i.e.* a term of the form **(that *wff1*)** where *wff1* is a wff, or a constant of sort **propos** such that an assertion about that proposition's content, *e.g.* `((lrrh girl) _ p1_propos)`, has been previously been made.

Simulative inference is performed during both entry and evaluation of belief literals. Given a literal **(*t1* believe *t2*)**, the system temporarily assumes as its own beliefs all beliefs previously attributed to *t1*, and then calls the **story** function (if the literal is being entered) or the **question** function (if the

literal is being evaluated) on the content of *t2*. Any formulas derived by input-driven inference in the simulation are thenceforth taken to be believed by *t1*.

5.14.1 Using the Belief Specialist

To start the belief specialist, issue the following command:

(use-spec 'belief-specialist)

The sorts used by the belief specialist are:

propos - the object of the verb "believe" should be of this sort.

The domain of the belief specialist consists of literals with the following pattern:

Argument1	**Predicate**	**Argument2**
Sort		**Sort**
	believe	*propos*

In the above pattern, the first argument may be of any sort.

5.14.1.1 Belief Specialist Display and Control

The belief specialist stores one environment for each agent about whose beliefs it has some information. The environment associated with an agent can be retrieved using the function **simulation-env**. The contents of a simulation environment can be examined using the same functions used for examining the system's own reasoning environment, by binding the variable ***environment*** to the environment in question. For example,

```
(let ((*environment* (belief:simulation-env 'lrrh)))
  (display 'wffs))
```

will display the wffs that the system believes LRRH believes.

In addition to the simulation environments, the belief specialist creates a shared environment that is used both by the system itself, and by all simulations. This environment is stored in the variable *shared-kb*.

The belief specialist has two tweakable parameters:

simulation-effort-level determines how much effort should be expended in trying to answer simulative queries. The value of this parameter is passed to the **question** function as its effort level. This is initially set to 3.

max-simulation-depth determines the maximum allowed depth of simulation nesting. Without this parameter, there would be the possibility of the system running a simulation, and the simulation running a simulation, and that simulation running another simulation, *ad infinitum*. This is initially set to 3.

Trace values for the belief specialist are:

belief-enter - Traces entry of literals into the belief specialist

belief-evaluate - Traces evaluation of literals by the belief specialist

belief-details - Traces details of belief specialist operation.

belief-all - Equivalent to **belief-enter, belief-evaluate, belief-details**

5.14.2 Details of the Belief Specialist

Details on the theory and implementation of simulative reasoning can be found in Aaron Kaplan's Ph.D. thesis (University of Rochester Computer Science Department, 2000).

The code of the belief specialist itself is very simple. It temporarily binds the global variable ***environment*** to the reasoning environment associated with the agent being simulated, and introduces a new scope for each of the global variables that **EPILOG** uses for storing inference state (so that a simulative assertion or query will not disturb a "real," *i.e.* top-level, assertion or query of which it is a part), and then calls the **question** or **story** function as appropriate.

The complexity of the belief specialist is in the requirements it places on the rest of the system. If you plan to add a new specialist to the system, and you want to be able to use both your specialist and the belief specialist, then your specialist knowledge representations and inference methods will have to satisfy these requirements. See Section 5.17 for instructions on creating new specialists.

5.15 Meta Specialist

The meta specialist is a specialist designed to enable some of the more complex, procedural testing required when trying to make inferences with meaning postulates. Literals belonging to the domain of this specialist are those involving "meta" information - information about a predicate, or a formula, rather than a specific individual. When a proposition is asserted or questioned in **EPILOG** which is in the domain of the meta specialist, the specialist is invoked. For example, asserting

(kill %action-pred)

would have *%action-pred* added to *kill*'s property list.

*((To (L x (L e1 ((x eat lrrh) ** e1)))) %action-type)*

would be evaluated to YES.

Note that some of the functions and predicates handled by the meta-specialist all begin with % - this is to differentiate them from regular object-level functions and predicates. The meta functions and predicates beginning with should only be used in meta assertions, meaning postulates and simplification schemas.

5.15.1 Using the Meta Specialist

The meta specialist is automatically activated when EPILOG starts up.

5.15.1.1 Meta Specialist Predicates

Predicates operating on one argument:

Predicate	Meaning (a1 predicate)
action-type	*a1* is an action term
%formula	*a1* is a wff
%action-formula	*a1* is an action formula
%action-pred	*a1* is an action predicate
%type-predicate	*a1* is a type predicate
%numeral-term	*a1* is a numeral term (number)
%episodic-term	*a1* is an episodic term
%kind-of-episode	*a1* is a term of the form (Ke ...)
%kind-level	*a1* is a kind-level predicate
%object-level	*a1* is an object-level predicate
%stative	*a1* is stative
%telic	*a1* is telic
%nonvolitional	*a1* is nonvolitional
%unlocated	*a1* is unlocated
%*-or-**	*a1* is either * or **
%atemporal-formula	*a1* is an atemporal formula
%currently-I-cannot-infer-likely	*a1* cannot be proven in a small inference attempt

Predicates which operate on more than one argument:

Formula	Meaning
(*wff* **%contains** *object* **)**	*wff* contains *object* in it somewhere

It is easy to add new meta routines - just update the file *meta/definition.lisp* , and add the new routines following the pattern of those in *meta/eval.lisp* . Note however - all new "meta" predicates should also be classified under the topic *tp.meta* .

5.15.1.2 Meta Specialist Functions

(%subst *new old wff*) [*function*]

This function returns a new normalized formula which consists of *wff* with all instances of *old* replaced by *new* .

5.15.2 Meta Specialist Display and Control

Trace values for the meta-specialist:

meta-eval - Traces evaluation of literals involving meta level objects

meta-function-eval - Traces evaluation of functions involving meta level objects

meta-entry - Traces entry of information into meta-specialist

meta-all - Traces all meta operations

The parts of the meta specialist that interact with the theorem prover can be controlled by the ***spec-enter*** and ***spec-evaluate*** flags (just like most other specialists). For minimal use of the meta specialist, these flags should be on.

5.15.2.1 Details of the Meta Specialist

If a literal involving one of the predicates recognized by the *meta-specialist* is asserted, has only one argument, and that argument is a symbol, the predicate is stored as a property on the property list of the argument - assertions involving non-symbolic arguments are not saved by the specialist.

During the evaluation phase, if the argument is a symbolic argument, the property list is checked first. If there is a property with the name of the predicate, YES is returned. If this fails, or if the argument is not a symbolic one, then the appropriate routine in the meta specialist (having the same name as the predicate) is invoked and used to do the evaluation. There are no restrictions on how this routine should work - it may be a completely procedural definition of the predicate, or it may check for certain properties, or work in whatever manner is required to evaluate the literal given to it. The routines may be recursive and may call on any routines in the rest of the system that they find useful. If there is no such routine, a default routine will check the argument for the property - this allows lambda expressions to be checked as well.

5.16 Other Specialist (Adding External Routines)

It may be desirable to be able to interface **EPILOG** with other Lisp programs and routines which do various kinds of information storage. For example, one might have a database of some special kind and want the assertions to **EPILOG** to be able to update this database, and want information from the database also available to assist **EPILOG** in evaluating and simplifying literals. This is easy enough to do if one sets up another specialist, but this may require more effort and a deeper understanding of the system than is desirable for such a (seemingly) simple task. An interface specialist, called the *other-specialist* has been set up which attempts to make this interface problem easier. The specialist just takes the names of predicates and functions that the user give it, the package where they can be found, and automatically calls these routines where appropriate in the system.

NOTE: if a modal embedding is involved, the routines made accessible through this specialist will NOT be called, as no provision has been made for handling different subnets. If it is desirable to interface extenal routines for modally embedded facts, a separate specialist will have to be set up. Negations are also not sent to the external routines.

In the event that this interface does not have the desired effect, it can be used as a pattern by which to build individual specialists which interface to the particular desired routines.

The commands needed to use this interface are as follows: Note: All routines which are called are responsible for checking the arguments themselves, and should not bomb if a different number or kind of argument is recieved than expected. Also note that the arguments are interned in the package *epilog* , so take this into consideration when testing or saving information. The "other" specialist ignores all modal embedding and subnet information - so be careful to use it only for predicates, etc which either will not be embedded in modal contexts, or for which the modal embedding does not matter. If it is important to take the modal embedding context into consideration, the global variable ***current-netname*** contains the current subnet name, and should be used for storing or extracting any modal context sensitive information.

(other-function *package &rest function-names*) [*function*]

Purpose: Tells the system that an evaluation function exists for a function and where to find it.

Syntax: *package* is the name of the package that the evaluation functions live in. Each name in *function-name** is a function which the user can use in formulas, and which will be evaluated when appropriate by an evaluation function in *package* by the same name.

Examples:

(other-function 'weather-pkg 'pressure-of 'temp-of)

would have *pressure-of* and *temp-of* created as functions in the system, and whenever they are used in a wff, the corresponding functions in the package *weather-pkg* will be called to evaluate them. So asking if *((pressure-of today) greater-than 50)*would cause the function *weather-pkg::pressure-of* to be called with argument *today* .

(other-predicate *package &rest predicates*) [*function*]

Purpose: Tells the system that an evaluation function exists for a predicate and where to find it.

Syntax: *package* is the name of the package that the evaluation functions live in. Each name in *predicate** is a function which the user can use in formulas, and which will be evaluated to yes, no, or unknown when appropriate by an evaluation function in *package* by the same name.

Examples:

(other-predicate 'weather-pkg 'stormy 'calm 'tornado-watch)

would create *stormy, calm* , and *tornado-watch* as predicates in the system, and would call their corresponding functions in the package *weather-pkg* to evaluate them (entry is also possible and will be discussed shortly). So if we question *(today stormy)* the routine *weather-pkg::stormy* will be called with argument *today* . This routine may use whatever method it likes to determine an answer. The answer it returns should be yes, no or unknown (in the *epilog* package), but the system can also operate with other output (non-nil is yes, nil can mean either unknown or nil, depending on the routines - the user controls this). The user defined routine will only ever be asked to answer positive predications - if the literal being evaluated is negated, the negation is stripped off, the positive literal is evaluated by the user's routine, and then the result is negated.

Note: if a routine returns a non-nil answer, it is considered true (unless it is *unknown*). If it returns nil, it is open to interpretation - this may mean *no* or *unknown* . There is a tweakble parameter ***interpret-nil-as-no*** which indicates which to use - if *t* , *nil* answers will be interpreted as *no* , otherwise as *unknown* . This is initally set to interpret *nil* answers as *unknown* . Remember that input facts are evaluated before entering, so one must be careful that the system doesn't think the input fact is false.

(other-entry-rtn *package predicate rtn*) [*function*]

Purpose: Tells the system that if an assertion is made involving a particular predicate, a corresponding entry routine exists to maintain that information elsewhere as well.

Syntax: *package* is the name of the package that the entry routine lives in. *Predicate* is a predicate which requires that assertions involving it be saved in some other form, handled by the routine *rtn* . Note, if *rtn* exists in *package* , that will be the routine used; if not, *rtn* is used as is.

Examples:

(other-entry-rtn 'weather-pkg 'calm 'set-calm-day)

If *(yesterday calm)* were asserted, the routine *weather-pkg::set-calm-day* would be called with argument *yesterday* .

5.17 Adding New Specialists

During propositional entry and question-answering, **EPILOG** can call specialists to assist it in several ways: *literal entry, literal evaluation, predicate comparison, literal comparison,* and *function evaluation* . This section describes the interface in more detail, and includes instructions on how new specialists can be added. The main system has a number of facilities set up to make this easier, and they should be used for consistency purposes as well. These include facilities for tracing, setting up tweakable parameters, setting up user display functions, and checkpointing and retraction of information. Details of some of the commands mentioned may be found in the "Library Routines" manual.

5.17.1 Requirements for the New Specialists

Before attempting to add a specialist, its domain must be determined, including what predicates are involved and what sorts of arguments it uses. Then the interactions it is to have with the main system must be decided upon. There are two kinds of specialists -

(*type1*) ones that simply maintain the relationships between predicates, and

(*type2*) ones that maintain their own separate representation of the portion of the facts in the story which fall into their domain.

The first kind can only be used to help compare predicates. This is the simplest and easiest kind of specialist to add, since one need only create a **compare-preds** routine (described shortly). The other kind has more possibilities for interaction, and at a minimum should have **enter** and **evaluate** interface routines. Literal comparison routines (**incompatible-lits, compatible-lits**) are optional, as are any functions usable in propositions. Note that for *type2* specialists, the representation and methods chosen must support checkpointing and retraction of information.

5.17.2 Steps to Adding a Specialist

This section gives the easiest order in which to do the necessary steps to add a specialist. Instructions and details on the steps are described in other sections. Note: not all steps will be required for all specialists.

1) Determine the domain of the new specialist
   Literal and function patterns must be decided on, and the exact effect they must have. Any sorts required must be defined, as well as function and predicate symbols.

2) Devise a special representation or method to do the inference
   Determine how this specialist will represent the information it handles, and how it will evaluate literals and functions. If possible, test these lower levels before incorporating into the system. For example, the type specialist uses a hierarchy with special numbering methods; the time and numbers specialists use graphs. If you want to be able to have the belief specialist and your new specialist active in the same system, then your representation will need to satisfy certain constraints, to be discussed below.

3) Start incorporating the specialist into the system
Create a directory for the specialist, and add the specialist to the file "specialists". Create a definition file (described below) for the specialist, and put it under this directory. Create the main load file for the system which exports the interface routines (don't export them yet). If there are any functions for this specialist, implement them first if possible, as it is easy to test them by asking a question

(q '((new-function ...) junk))

while tracing *function-eval* to see what happens.

4) Write the entry interface (*type2* specialists only)
Write the entry routine for the specialist to interface it with the system - the required definition will be described shortly. Export the symbol **enter** so that the interface knows it is available. Write some display routines so that you can see if it is entering things properly, and possibly a trace value as well. Any assertion the specialist wants to make back to EPILOG should be put on the list ***assertions*** .

5) Add the literal evaluation interface (*type2* specialists only)
Add the interface and any traces desired to do literal evaluation. Export the symbol **evaluate** so that the interface knows it is available. Test this out with the system. When satisfied that simple literal entry and evaluation, and function evaluation are working, continue with the next step.

6) Add predicate comparison (*type1* specialists only)
Export the symbol **compare-preds** . When called, this should return the relationship between the predicates given. The relationship returned should be one of *disjoint, subsumes, subsumed, equivalent, unknown* .

7) Add literal comparison (*type2* specialists only)
Export the symbols **incompatible-lits** and **compatible-lits** . See the section of literal comparison for some hints on how to do this. This is by far the most difficult, hardest to test, and least rewarding of the possible specialist functions.

5.17.3 Details

This section contains details on setup and interfaces for specialists.

5.17.3.1 Definition and File Setup

Each specialist is assumed to live in its own directory which should be in the *epi* directory (although for initial testing it may be elsewhere as long as the *loadfile* for **define-specialist** and *defn-file* for **set-specialist** are set properly). In that directory, there should be a file containing the definition of the specialist and any required initializations on start-up (this file is usually called *definition* for ease of finding it, but may be set to anything using the **set-specialist** command.

In this definition file should be a command of the form:

(define-specialist *specialist-name loadfile predicates operators sorts functions topic* **)**

This function is not called until the definition file is loaded, which takes place only when **use-spec** is called. This will define the specialist to the system, tell it where to load the specialist from, and note any predicates and functions it uses. If *loadfile* exists, it is **require** d. Each predicate in *predicates*

has *specialist* associated with it on its property list, and is marked as a predicate. The same is done for each function in *functions* except that the function is also marked as a function. *Sorts* is a list of sorts that are allowable for the first term in a literal for this specialist. If *topic* is specified, it will be used to topically classify formulas which involve predicates the specialist handles. They are stored for later use.

In addition, any functions which are to be transformed into relational predicate during assertions should be defined here using **function-with-rel-pred** . More on this in the section on functions ...

Each specialist must exist in a package of its own, named with the specialist name (eg. *time-specialist* with nickname *time*). The specialists must reside in separate packages, because they will have conflicting symbols (*enter, evaluate, compare-preds, incompatible-lits, compatible-lits* , plus possibly some functions as well).

The file *specialists* (same level as **EPILOG** .lisp) must be updated with information about the new specialist. Entries in this file are of the form:

(set-specialist *specialist-name defn-file nicknames description* **)**

Defn-file indicates the file that contains the definition for the specialist. The specialist is added to the available-specialists list (***available-specialists***), but is not activated until a **use-spec** command is issued.

In addition, the predicates involved in the specialist should be put into the same topic in the topic hierarchy, so that propositions using them will be classified under the same topic. Otherwise the propositions will never be found to compare against. If all predicates are known in advance, this can be done in the specialist definition. If not, the specialist itself can add the appropriate indicator topic to the predicates as they are given to it (the color specialist and type specialist do this).

5.17.3.2 Requirements Imposed by the Belief Specialist

If you want it to be possible for your specialist and the belief specialist both to be active in the system, then your specialist must satisfy certain additional requirements.

First, if your specialist can invoke other specialists via the immediate evaluation interface (see page 91, then it must be reentrant: your specialist could call the belief specialist, which could run a simulation, in which your specialist could be invoked again, nested within the original invocation. In particular, this means that you should avoid storing inference state in global variables. The belief specialist introduces a new scope, using **let**, for each of the existing system's global variables that contains inference state, but it doesn't know about any new variables you may introduce. If you must store inference state in a global variable, you will have to modify the belief specialist to handle it explicitly.

Second, if you are writing a type 2 specialist, *i.e.* one that has an **enter** function for storing information in some special-purpose data structure, then this data structure should be stored in the environment. That way, the system's own beliefs in your specialist's domain can differ from the beliefs the system attributes to others in that domain. The specialist interface provides the **retrieve-specialist-info** function to facilitate this:

(retrieve-specialist-info *environment specialist &optional init-function* **)**

specialist should be a symbol that names the specialist that's calling the function. You will typically use the current environment ***environment*** for the *environment argument*. The first time **retrieve-specialist-info** is called with a particular pair of values for *environment* and *specialist*,

the function *init-function* is called to create a new data structure of whatever type the specialist needs, and that object is stored in *environment* under the name *specialist*, and returned by **retrieve-specialist-info**. On subsequent calls, when *environment* already has something non-nil stored under the name *specialist*, that object is returned.

Third, to take full advantage of the facilities for simulative inference, the specialist should be able to use information from multiple environments simultaneously. There will typically be a large amount of world knowledge which the system believes, and which it also believes that everyone else believes. Since it would be unreasonable to duplicate all of this information in each reasoning environment, the belief specialist provides the **belief:*shared-kb*** environment and makes the information in it accessible from all other environments. Therefore, all inference methods, both at the top-level and in simulations, must be able to combine information from multiple environments. A specialist should never modify any environment other than the current one (stored in the variable ***environment***), but in answering queries it should be able to use information from the current environment as well as any environments accessible from it. The environments accessible from the current one are given by **(environment-use-environments *environment*)**. For a discussion of the difficulties of writing specialists that use multiple reasoning environments, and some techniques for coping with these problems, see (Kaplan 2000). Also see **EPILOG**'s set specialist, equality specialist, and time specialist as examples.

It might be that for your specialist domain, either all agents can be expected to have the same beliefs about that domain, or on the contrary the number of shared beliefs about the domain can be expected to be small compared to the number of non-shared beliefs. In these cases, it may not be worthwhile to implement multiple-environment reasoning. In the first case, you may choose to have the specialist use only the **belief:*shared-kb*** in both top-level reasoning and simulations, ignoring the current environment ***environment***. To avoid unexpected results, your specialist should modify the **belief:*shared-kb*** only during top-level reasoning, not during simulations. To this end, the variable **belief:*at-top-level*** is provided. Its value is **t** during top-level reasoning and **nil** during simulations. In the latter case, when is is expected that there will be few shared beliefs in the specialist's domain, you may choose to have the specialist use only information stored in the current environment ***environment***, ignoring environments accessible from that one.

5.17.3.3 Entry/Assertion of Literals

The specialist interface will call the following routines in the specialist when an asserted literal looks like it might be of interest to the specialist. **Enter** should be defined and exported. It is a good idea to have some trace values set up to watch the action of the specialist as it enters the information given to it in its own representation.

(enter *netname lit negp pred arglist* **)**

This routine will be called when the interface determines that this specialist may be interested in a literal being asserted. *Netname* is a symbol of the form */concept1/concept2 ...* indicating which subnet this literal is for (main is **/** , John's mental world is **/john** , etc). *Lit* is the literal within *netname* to be entered. *Negp* indicates whether or not it is negated. *Pred* is the basic predicate (operators are stripped off) of the literal, and *arglist* the arguments.

Enter is responsible for checking the arguments given to it to make sure they are of the correct sort, and are constants if that is a requirement of this specialist. If something was actually entered, it should return t, otherwise nil.

Once a literal has been asserted to all applicable specialists, each term is checked to see if it has an interested party list (see under Specialist Communication in Chapter 5). If so, all the literals on that list are also reasserted to the specialists that were interested in them.

A specialist may need additional information when it is entering a literal which may be supplied by other specialists. There are two types of information that may be requested - the evaluation of a functional term, or the evaluation of a literal. The specialists involved so far in this system only required the former. The specialist interface command **eval-fn** may be used to get the desired information. For example, the time specialist might be given literal *(e1 has-duration d)* , and want to know the bounds on *d* , so it would request *(eval-fn (list 'max-of d))* , and *(eval-fn (list 'min-of d))* . These may or may not return an answer (depending on whether that information has been obtained yet). In addition, the current literal and specialist should be added to the interested party list for any such concept that it tries to get information for, For the time specialist in the above example, bounds on *d* may be asserted at a later time, and the time specialist would then be invoked with that same literal again. This time it would get different results for the function evaluations. **Add-interested-party** should be used to add to a concept's interested party list.

During the entry of a literal, information may change for a constant which was not in this literal (via propagation for example). This may lead to additional inferences in other specialists. A specialist may indicate to the specialist interface that it has changed something (using the command **spec-changed-concept**). All the concepts indicated in this manner are put onto a list, and after the current literal has been asserted to all the interested specialists, and the related interested party lists have been re-asserted, the interested party lists for each of these concepts is also reasserted (Note - there is a possibility for infinite looping here if a specialist puts the concepts in the current literal on this list, or if it puts concepts on that have not actually changed).

A specialist may be able to make input-driven inferences based on the literals given to it. If these inferences are placed on the special list ***assertions*** , EPILOG will find and assert them (as long as ***spec-assert*** is t), after all the specialists have entered the current literal, and before any other input-driven inference is attempted. These assertions must be in unnormalized, list form.

5.17.3.4 Evaluation of Literals

The specialist interface will call the following routine in the specialist when a literal being evaluated looks like it might be answered by the specialist. **Evaluate** should be defined and exported. It is a good idea to have some trace values set up to watch the action of the specialist as it evaluates the literal given to it.

(evaluate *netname lit negp pred arglist effort* **)**

This routine will be called when the interface determines that this specialist may be able to evaluate a literal. *Netname* is a symbol of the form */concept1/concept2 ...* indicating which subnet this literal is for. *Lit* is the literal within *netname* to be evaluated. *Negp* indicates whether or not it is negated. *Pred* is the basic predicate (operators are stripped off) of the literal, and *arglist* the arguments. **Evaluate** is responsible for checking the arguments given to it to make sure they are of the correct sort, and are constants if that is a requirement of this specialist. The answer returned should be YES, NO or UNKNOWN.

As in the entry of literals, sometimes the specialist may not have all the information it needs, and may want to request the evaluation of a functional term or literal. **Eval-fn** can again be used here. The

interested party lists are of no use here, since evaluations are fleeting and later information will be too late to help this evaluation.

5.17.3.5 Comparison of Predicates

The following routine is called when the specialist interface decides that the specialist could be used to determine the relationship between two predicates. **Compare-preds** should be defined and exported. Note, for a *type1* specialist, this is (usually) the only interaction between the specialist and the main theorem prover.

(compare-preds *pred-1 pred-2* **)**
Pred-1 and *pred-2* are the predicates to be compared, including operators. The specialist is responsible for examining any top level operators to make the final decision about whether the predicates are in its domain. Most specialists will either ignore the operators or discard predicates with operators, except for the color specialist, which takes into account several operators, including *sort-of* and *almost* .

5.17.3.6 Comparison of Literals

There are two comparison routines that may be set up in a specialist for interfacing with the main system. These are **incompatible-lits** and **compatible-lits** , which should be defined and exported (note that these routines are optional). The specialist interface will call them when it has two literals to be tested for incompatibility/compatibility which both trigger this specialist.

(incompatible-lits *netname lit-1 lit-2 neg-p1 pred1 arglist1 neg-p2 pred2 arglist2 effort* **)**
This routine is called when the interface determines that the specialist might be able to tell if *lit-1* and *lit-2* are incompatible. Both literals are from the same subnet, indicated by *netname Neg-p1* indicates whether \m lit-1 is negated. *Pred1* is the basic predicate (operators are removed) for *lit-1* and *arglist1* its arguments. Similarly, *neg-p2 pred2* and *arglist2* apply to *lit-2* The specialist must unify the argument lists and return a list of "agenda items" with the appropriate substitutions and residues. This process and the order of unification depends on the predicates involved and whether or not the literals are negated. There may be more than one possible action which results.

(compatible-lits *netname lit-1 lit-2 neg-p1 pred1 arglist1 neg-p2 pred2 arglist2 effort* **)**
This routine is exactly like **incompatible-lits** except that it determines whether the literals are compatible.

Note that **eval-fn** may be used to request information as in the entry and evaluation interfaces.

These are by far the hardest functions to ask the specialist to do. For some specialists, this testing can be done in the following manner:

a. Decide what order to unify the arguments in

b. For each unification (use **unify-lists** or **compare-arglists**):

   i. Make the substitutions in the two literals
      - use **substitute-prop**

ii. Simplify the arguments in the two literals
- use **getargs** to get the simplified arguments, and **arg-sub-list** to determine those substitutions so that they may be added to the substitution lists

iii. Evaluate both literals
- use the evaluation routine developed for this specialist. Use the routine **incompatible-with-eval** to determine if this is enough - if it returns a non-nil answer this attempt is finished. Otherwise continue ...

iv. Checkpoint
- to ensure that you don't make any permanent changes here

v. Enter one literal into the specialists representation

vi. Evaluate the other literal
You can either use the evaluation to determine compatibility/incompatibility, or use it to determine the conditions necessary to make the literals incompatible/compatible. In the later case, the condition must be returned as a *residue* - a proposition.

vii. Retract changes made from last checkpoint

5.17.3.7 Functions

In addition, any number of functions may be defined for use in propositions. These functions should all be defined with an argument list of *&rest args* (for an example, see the end of this section), and should NOT abend if the number of arguments is incorrect. If the arguments are inappropriate, an answer of nil should be returned; otherwise a concept or quoted expression with the result of the function application should be returned. To remain consistent, they should also test to see if *function-eval* is being traced, and if so, print the result of the function before returning.

Note - it may be desirable to have some functions able to set information on assertion (normally the functions are evaluated only). For example, the time specialist has a function **duration-of** which can be used to set an event's duration, and similarly, the set specialist has a function **cardinality-of** which can be used to assert the cardinality of a set. To do this, the specialist interface determines the relational predicate which can be used to achieve this, and creates a new unique constant of the sort that the function normally produces, and produces several assertions to be made to the specialists (this does not go back to the main system!). For example:

((duration-of e1_episode) less-than (duration-of e2_episode))

would be transformed into the following:

(d1_number less-than d2_number)

(e1_episode has-duration d1_number)

(e2_episode has-duration d2_number)

To accomplish this, a predicate must first be set up which uses the same information, and actually sets the desired fields. This predicate is then tied to the function using the command

(function-with-rel-pred *function-name predicate-name sort* **)**

where *sort* is optional and is the sort of result that the function produces (when evaluated). This command should be placed in the *definition* file for the sepcialist after the **define-specialist** command. (If desired, this command may be used simply to add a sort to the function by invoking it with a null *predicate-name*).

5.17.3.8 Notes

For any of the interface functions (including the functions usable in propositions), when passing the names of concepts (the predicate or arguments) around, be aware that their package names will be *epilog* , so any comparisons must have *epilog::* in front. Alternatively, the equivalent symbol in the specialist's package may be found and used inside the specialist.

Any changes made to the specialists representation by these routines should be added to the checkpoint stack using *checkpoint* . When changing properties, hash table entries, array entries, etc, use the function **change-property** , **change-field** , **change-hash** and **change-array** to do the changes - this automatically sets up the retraction list so that if a retract is called for, everything is returned to its previous state. It is possible to set up your own retraction routines, if there is a better way to do this than by saving atomic changes. See the "Library Routines" manual for details.

For ease of implementation and consistency, note the following. For parameters which may be set that are specific to this specialist, use **tweakable** . To allow the user to display information contained in the specialist, use **set-display-function** . Use **prinlis** for printing display information and error messages. To allow the user to trace certain aspects of the specialists execution, use **traceable** to set up new trace values, and **traced-p** to determine if a symbol is being traced. These routines are described in the "Library Routines" manual. Note: all trace output should be prefixed by 4 spaces and the name of the specialist. For example, *" Time Specialist: Evaluating ... "*

5.17.3.9 Documentation

All new specialists should be documented in this manual. For consistency, user information (information on how to use the specialist - what predicates and arguments it takes, tweak and display functions, etc) should be placed in Chapter 5 (Specialists) in a subsection for the new specialist. Information should be added in the following order (note that not all of these will be needed for every specialist): introduction, sorts used/defined for this specialist, predicates/operators involved, functions defined, display functions and controls, and details. The "Quick Reference Guide" and "Programmer's Guide" should also be updated.

5.17.4 Example Specialist

The following example shows how the temporal specialist package is set up:

```
(provide 'time-specialist)
(in-package 'time-specialist :nicknames '(time) :use '(lisp user epilog))
(export '(
        ;; These are optional but must be exported to be used
        enter evaluate incompatible-lits compatible-lits

        ;; any functions which may be used in propositions
        start-of end-of date relation duration elapsed

        ;; Note that display, tweak, rewind and unifiable functions are not
        ;; required to be exported, as their internal names are passed directly
        ;; to the net.
))
```

```
(traceable 'epilog::time-entry "Traces input of temporal relations")

(defun enter (netname lit negp pred arglist)
       ; enter in specialized time graph
       t
  ...)
(defun evaluate (netname lit negp pred arglist effort)
       ; look in specialized time graph
       (cond (true-literal 'YES)
             (false-literal 'NO)
             (t 'UNKNOWN))
   ...)

(defun unifiable (arg1 arg2)
       ; see if worth trying any comparisons with these
   ...)

(defun incompatible-lits (netname lit-1 lit-2 neg-p1 pred1 arglist1
                          neg-p2 pred2 arglist2 effort)
   ...
  ; determine order(s) of unification
  ; for each order:
  (unify-lists arglist1 arglist2 ... )
  (list
       (make-comparison-info :residue1 ... :subs1 ... :residue2 ... :subs2 ...))
       ; return as many different incompatible results in this list as
       ; are found with the different substitutions
  ...)

(defun compatible-lits (netname lit-1 lit-2 neg-p1 pred1 arglist1
                          neg-p2 pred2 arglist2 effort)
   ...
  ; determine order(s) of unification
  ; for each order:
  (unify-lists arglist1 arglist2 ... )
  (list
       (make-comparison-info :residue1 ... :subs1 ... :residue2 ... :subs2 ...))
       ; return as many different compatible results in this list as
       ; are found with the different substitutions
  ...)

(defun start-of (\&rest args)
    (unless (null (car args))
            find and return concept for start
    ))

... other functions
```

Chapter 6

Response Generation

EPILOG has a natural language generation facility which can give a more helpful and fuller answer to a question than just a simple yes or no, or can be use to "say" input facts or groups of facts. This can be useful to help detect errors in input, as the English response will try to capture the meaning of the input. These response functions are automatically turned on, but may be turned off if so desired. Translation and lexical information may be given to the system to make it say things in a more natural manner - this information is described in Chapter 4.

The response generator tries to build a language fragment, or set of fragments for each wff given to it (using the translation information given to it), and then combines those fragments using some heuristics. More heuristics and some knowledge base lookup are used to fill in any "gaps" in the resulting sentence fragment(s) to get a complete and meaningful sentence. The response generator also tries to use pronouns where applicable, and shortened descriptions of referring noun phrases. Currently the generator can handle most of the allowable syntax fairly well, but sometimes gets confused with complex formulas, and combinations of formulas.

6.1 Using the Response Generator

Optionally, the system will automatically generate English output for answers to questions (if flag ***say-answer*** is set), input knowlege (***say-knowledge***), story information (***say-story***) and input-driven inferences (***say-infer***), meaning-postulates (***say-mp***) and meta information (***say-meta***). The English generated for meaning postulates and meta information is not particularly good and the system is easily confused, so these are set off by default. In addition, there are some commands (**do-say, say-it, say-them**) which can be used to "say" things on demand.

This section describes how to input lexical and translation information, the commands for invoking the response generator, and the various controls for the response generator.

6.1.1 Response Generator Commands

In addition to the commands below, there are two other commands: **add-word** for adding translation information, and **add-lex** for adding lexical information. These are described in the sections *Translation Information* and *Lexical Information* , respectively.

(do-say *arguments*) [*function*]

Purpose: To repeat an answer (optionally in more detail), or to say information known about a specific concept and topic.

Syntax: The first element of *arguments* is a flag which may be one of:

answer or **ans**
the answer to the last question is repeated.

more-details or **more** or **details**
the answer to the last question is repeated in more detail, by using the parents of the original wffs in the answer. This may be repeated until the wffs have no parents, and then just a Yes or No will be returned.

less-details or **less**
just the reverse of the above process. This will just repeat the answer unless a *do-say 'more* has been done, in which case it will repeat the previous version of the answer.

retrieve or **ret**
The rest of *arguments* is used as the arguments to the **retrieve** command, and the results of the retrieval are then said.

a list of formuals
The ***prompt-if-too-complex*** flag is temporarily set to *t* and the list of formulas is passed to **say-it** .

Examples:

(do-say 'ans)

(do-say 'more)

(do-say 'ret 'lrrh)

(do-say 'ret 'wolf 'tp.coloring)

Remarks: Note that **do-say** re-filters each time it is asked to print the answer. If desired, the filter threshold (***filter-threshold**) can be changed and a question answer "resaid" this way without having to go through the question answering mechanism again.

(say-it *list-of-wff*) [*function*]

Purpose: To say an arbitrary group of wffs as a sentence.

Examples:

(say-it '(wff1 wff10 wff34))

Remarks: Be careful about what groups of wffs you decide to say - some do not combine nicely and the system is not good at detecting this (for example, combining *LRRH smaller than bed* and *LRRH not smaller than basket* can give incomplete and strange sounding sentences.

(say-them *list-of-lists*) [*function*]

Purpose: To say a set of sentences, each made up of a group of wffs, as a coherent whole.

Examples:

(say-them '((wff1 wff10 wff34) (wff2 wff3) (wff5)))

Remarks: Pronouns are used across sentences to give coherence. The same comment goes for these groups of wffs as for the wffs in **say-it** - be careful what you decide should be said together.

6.1.2 Response Generator Display and Controls

The following are display functions that can be called through the display command (all arguments are optional):

(display 'pred-info *preds* **)**
For each predicate in *preds* , the predicate's type and translation as used for response generation are displayed.

(display 'lex-info *words* **)**
For each word in *words* , the word type and lexical information as used for response generation are displayed.

The following tweakable parameters may be used to control the actions of the response generator:

say-answer
which determines whether or not the answer obtained using the question answering mechanism should be said in English. This is initially set to *t* but may be changed using the **tweak** command.

say-knowledge
which determines whether or not to try to "say" input knowledge (rules). This is initially set to *t* but may be changed using the **tweak** command.

say-story
which determines whether or not to try to "say" story knowledge. This is initially set to *t* but may be changed using the **tweak** command.

say-infer
which determines whether or not to try to "say" inferred (input-driven) facts. This is initially set to *t* but may be changed using the **tweak** command.

say-mp
which determines whether or not to try to "say" input meaning postulates. This is initially set to *nil* but may be changed using the **tweak** command. The system doesn't say meaning postulates very well.

say-meta
which determines whether or not to try to "say" input meta information. This is initially set to *nil* but may be changed using the **tweak** command. The system doesn't say meaning postulates very well.

say-question
which determines whether or not to try to "say" the question before trying to answer it. Currently the question is "said" as a statement - a future enhancement will be to ask it as a question. This is initially set to *nil* but may be changed using the **tweak** command.

say-immediate
which determines whether or not to try to "say" each partial piece of knowledge (inferred, story or rules) as they are entered. If an input fact may be split into several wffs, they are usually gathered together and said as one. With this flag, however, the pieces are said individually, as they are entered. The flag is initially set to *nil* but may be changed using the **tweak** command, although this should be done only for testing - the other methods give more natural sounding English.

default-lex
If this flag is set, instead of prompting the user for lexical information, the system uses its default rules to figure it out for itself. This flag is initally set to *t* , but may be changed using the **tweak** command.

default-trans
If this flag is set, instead of prompting the user for translation information, the system uses its default rules to figure it out for itself. This flag is initally set to *t* , but may be changed using the **tweak** command. If the system uses a default, the user will be warned.

filter-threshold
When filtering, all set-of-support clauses are automatically filtered out. Then all clauses with a likelihood greater than the filter-threshold are filtered out (unless there is only one clause left). The filtration threshold is initially set to 50, but may be modified to any number between 0 and 100 using the tweak command. At 100, nothing except set-of-support clauses are filtered out; at 0, only the clause with the lowest likelihood of being known will be used.

max-response-complexity
This may be either nil, which indicates that all formulas should be attempted by the response generator, regardless of complexity, or a number which is the maximum complexity to attempt to say. The default is 40.

prompt-if-too-complex
If this flag is t, when a formula is detected which is too complex to say (using ***max-response-complexity*** , the user will be prompted to see if he wants the system to attempt it anyway. If nil (the default), the formula will just be ignored.

response-warn
This flag indicates whether or not to print response generation warnings. It is initially set to t, so that warnings about possible strange sounding output will be printed.

active-topics
This parameter is used by the response generator to help determine how to say certain predicates when no translation information is available for them. For a predicate with only one argument, it is considered a *noun* is the predicate is a type predicate, a *verb* if one of the indicators for the predicate is on the ***active-topics*** list (or is beneath in the topic hierarchy), and an *adjective* otherwise.

Trace values for the response generator are:

filtration - Shows which formulas were filtered out and why.

filtration-details - Shows the estimation of the likelihood of a clause being known.

response - Displays the wffs being input to the response generator (after filtration).

verbalization - shows the set of wffs used for each sentence, and the sentence fragments resulting from that set.

fragment - shows the creation of fragments for clauses and literals.

combine - shows how the fragments are combined.

combine-details - shows more detailed information on all combining attempts.

retrieval - shows any wffs taken from the knowledge base to be used in filling in a fragment.

trans-details - shows the details of translating into fragments (for debugging)

response-all - shows everything about response generation (except *trans-details*)

response-int - shows interesting things about response generation (including *filtration, organization, verbalization, response, fragment,* and *retrieval*).

response-min - shows minimum things about response generation (including *filtration, verbalization, response,* and *retrieval*).

6.1.3 Translation Information

All predicates are assumed to represent one of the following English forms: singular noun, verb infinitive, adjective, or preposition. To enter translation information about a predicate (for use in creating fragments), the command **add-word** is used. The format of the command is as follows:

(add-word *pred translation-list*) [*function*]

Purpose: To describe a specific bit of grammar for use in translating formulas with that particular predicate.

Syntax: *pred* is the predicate to be defined. *translation-list* is a list of the form

({fragment-type} {number} {property})*

where either *fragment-type* or *number* must be specified.

property -> (property-name item+)

item -> translation-list | symbol | number | (special-routine args)

The grammar used by the system is shown in the Details chapter, and this "bit" of grammar should be consistent with it. *Fragment-type* must be one of *S, NP, VP, PP, AP, ADVP, ADVS* . *Property-name* is either a fragment type (to describe a fragment within a fragment, like the NP for an S), a group fragment type (*parts or objects*), or a simple property name (*number, tense, aspect, noun, adj, adv, particles, pre ,conj, name, required*). The best way to set up translation information for a new predicate is to examine the translation information for similar predicates. The particular properties used in *translation-list* differ for each possible "type" of predicate.

Examples:

verb-like predicates: use properties *verb aspect tense objects adv particles*

(add-word 'sleep-in '(S (NP 1) (VP (verb sleep) (objects (PP (prep in) (NP 2))))))
makes a prepositional phrase after *sleep* for the object being slept in (*Little Red Riding Hood slept in a bed*).

(add-word 'blow-up '(S (NP 1) (VP (verb blow) (particles up))))
shows the use of particles (*A bomb blew up*).

(add-word 'eat '(S (NP 1) (VP (verb eat) (objects (NP 2 (required nil))))))
makes the object of *eat* a noun phrase, which may be omitted if there is no information about it. It is ok to say "John eats" but not ok to say "John lives in". (*The wolf ate Little Red Riding Hood* , *Only living things eat*).

(add-word 'say '(S (name 1) (VP (verb say) (objects 2))))
lets the object of *say* be whatever the particular term works out to be (an S, NP, etc) - *John said that Mary loved Bill* .

(add-word 'like-to-eat '(S (name 1) (VP (verb like) (objects (VP (verb eat) (aspect infinitive) (objects (NP 2 (required nil))))))))
E.g. *Wolves like to eat creatures* . Note that constructions like this can also be handled by using lambda abstractions in the logic (e.g. *(wolf1 like (To (L x (x eat ...))))*).

noun-like predicates: use properties *noun number PP AP S*

(add-word 'living-thing '(NP (name 1) (noun thing) (AP (adj living))))
describes this compound predicate as a combination of a noun and an adjective (*Only living things eat*).

(add-word 'body-of '(NP (name 1) (noun body) (PP (prep of) (NP 2))))
describes this compound predicate as a combination of a noun and a prepositional phrase (*C1 is a body of Little Red Riding Hood* - usually transformed into *Little Red Riding Hood has a body*).

preposition-like predicates: use properties *NP S prep pre*

(add-word 'before '(S (name 1) (ADVP (prep before) (S 2))))
makes a *before* predication into an adverbial phrase modifying a sentence (*The wolf ate Grandmother before it ate Little Red Riding Hood*).

(add-word 'greater-than '(PP (name 1) (pre more) (prep than) (NP 2)))
The cardinality of the set of wolves is more than 1 .

adjective-like predicates: use properties *adj adv*

(add-word 'pretty '(AP (name 1) (adj pretty)))
Little Red Riding Hood is pretty .

modifying operators/functions: use any of the properties
If the operator produces another operator, the arguments will be in the order of innermost operator to outermost (see *coll-of*).

(add-word 'very '(1 (adv very)))
Very is an intensifier which acts on an adjective. E.g. *Little Red Riding Hood is very pretty* .

(add-word 'coll '(NP (noun group) (number singular) (PP (prep of) (NP 1 (name removed) (number plural)))))
Coll creates a noun phrase from a predicate. E.g. *There is a collection of wolves .*

(add-word 'coll-of '(NP (noun group) (number singular) (PP (prep of) (NP 2 (number plural) (determiner 1)))))
Coll creates a noun phrase from a predicate. E.g. *There is a collection of wolves .*

(add-word 'ly '(VP 2 (adv (make-adverb 1))))
Ly makes an adverb from a predicate by adding *ly* to the end. E.g. *Little Red Riding Hood quickly walked .* The *make-adverb* is a user-defined routine for making an adverb out of whatever argument is sent to it. The mechanism for adding these is in place but is not completely defined or tested - instructions on using it will be available at a later time.

(add-word 'to '(VP 1 (aspect infinitive)))
To makes a term out of a predicate. E.g. *The wolf wanted to eat Little Red Riding Hood .*

(add-word 'start-of '(NP (determiner "the") (noun start) (PP (prep of) (NP 1))))
Start-of is a function. Since there can be only one start for an episode, the determiner "the" has been added (don't worry about using strings in some places and atoms elsewhere - the system can handle both). E.g. *The start of E1 is before E3.*

Remarks: Note that all functions MUST have the property *function* on their property list. Specialists automatically add this property to their functions when activated, but user defined functions must have it added manually. Otherwise the system will not be able to normalize or classify the wff properly, or say it in a natural sounding way.

6.1.3.1 Translation Defaults

If no explicit translation exists for a predicate, the system will try to guess in some cases. If the flag ***default-trans*** is off, it will prompt the user for the information; otherwise it defaults as follows:

A literal with only one argument - the predicate will be assumed to be a **noun** if the predicate is on a type hierarchy; a **verb** if one of its topic indicators is on the ***active-topics*** list; otherwise it is assumed to be an **adj** ective. (translations *(NP (name 1) (noun predicate))* , *(S (NP 1) (VP (verb predicate)))* , and *(AP (name 1) (adj predicate))* respectively).

If there is more than one argument, the predicate is assumed to be a **verb** , with arguments as simple numbers - as many as there are arguments after the subject. (e.g. *(S (NP 1) (VP (verb predicate) (objects 2 3)))*)

The defaults (both translation and lexical) are designed to avoid having to make up huge files of translation and lexical information, and to allow tests with new predicates with a minimum of frustration. Using the lexical defaults all the time is recommended, as the worst that can happen is one word will come out with an incorrect form. The translation defaults are (unfortunatly) usually wrong however, since they have very little information to work with, and the sentences will come out very strange if new words are used (especially if they are prepositions). The advantage of using the defaults is that it is easier to add new predicates without having to worry how they should be said. Translation defaults are saved on the predicate or operator they were calculated for to save time in future responses. Lexical defaults are not saved.

6.1.4 Lexical Information

To enter lexical information, the command **add-word lex** is used. Note – lexical entries are strings and should be entered in quotes. Note that not all information needs to be entered - if the ***default-lex*** flag is true, the system can sometimes figure out what the form should be.

(add-lex *word type key-entries*) [*function*]

Purpose: To add lexical information.

Syntax: *word* is the word to be entered into the lexicon. *type* is one of: **verb noun adj name other** *key-entries* is a set of keyword parameters and depends on *type* . These are described below. Note that both *word* and *type* are required. To make it easier to add lexical inforamtion, **add-lex** accepts a number of keyword parameters - some for each type of word.

verb: *:present :past :negative-present :negative-past :passive :pres-part*

All verbs must be entered as their infinitive (so *word* is the verb infinitive without "to"). *Pres-part* and *passive* represent the present participle and passive forms. If *passive* is **'no-passive** , it means that the verb cannot be passivized, so it will not try to default the passive form. *present* is a list of two forms - 3rd person present singular, and 3rd person present plural. Similarly, *past* is also a list of two past forms - one for singular, the other for plural. The *negative-present* and *negative-past* are the corresponding negated forms. For example, the verb *do* has the following lexical information

(add-lex 'do 'VERB :present '("does" "do") :past '("did" "did")

:negative-present '("does not" "do not")

:negative-past '("did not" "did not")

:passive 'no-passive)

Note that the *pres-part* key was not filled in - in that respect the verb is regular, and we can use the system default for it (*"doing"*).

noun: *:plural :props*

The singular form of the noun is the string form of *word* . *plural* is the plural form, and *(properties)* is optional and consists of a list of properties of that noun that might be useful (recognized properties are on the list ***lexical-properties*** - which consists of *mass, feminine, masculine, group,* and *member*) Note that a plural form is not necessary for mass nouns. For example, the lexical entry for *honey* would be

(add-lex 'honey 'NOUN :props '(mass))

while *wolf* would be

(add-lex 'wolf 'NOUN :plural "wolves")

The entry for *woman* is

(add-lex 'woman 'NOUN :plural "women" :props '(feminine))

Note that the system can default on the plural form of the noun so it need not be entered if it is regular.

Currently the system looks for a property of *mass* when deciding on number and determiners. *Feminine* and *masculine* are used to determine pronouns to use when referring to a noun phrase that has been said already. The *group* and *member* properties are used to help the system determine

when not to say prepositional phrases with *of* . For example, *all members of the group of wolves ate the steak* would be said as *the group of wolves ate the steak* (*member* property - initially *one* , and *member* have this property). And *The office was full of a group of computers* would be said as *The office was full of computers* (the *group* property - initially *group, set* and *collection* have this property).

name: *:trans :subject-pronoun :object-pronoun :props*
where *trans* is the string version of the name, the pronouns are optional, and *props* is exactly like the properties for *noun* s. For example, the lexical entry for *LRRH* is

(add-lex 'lrrh 'NAME :trans "Little Red Riding Hood"
:subject-pronoun "she" :object-pronoun "her")

and for *John* ,

(add-lex 'john 'NAME :trans "John" :props '(masculine))

adj: *:trans*
where *trans* is the string form of the adjective. This defaults to the string form of *word* , with - removed. For example, *pale-skinned* could have lexical entry

(add-lex 'pale-skinned :trans "pale skinned")

although the default would be that anyway.

other: *:trans*
where *:trans* is the string form of the preposition, article, etc (treated the same as adjectives).

If any lexical information about nouns or verbs is missing, the system will ask for the particular form it needs (singular/plural, past/present, etc). Note that these should always be entered in double quotes (" "). Adjectives, articles, etc, will default to the string form of their name.

6.1.4.1 Lexical Defaults

If the ***default-lex*** parameter is turned on, instead of prompting the user for a missing lexical entry, the system will just "build" it. These are the rules it uses:

Verbs:

3rd person present singular: adds "s" to the verb infinitive (e.g. *he eats*). If the verb infinitive ends in "s", "x", "h", or "z", "es" is added instead (e.g. *he fixes*); if it ends in "y" and does not have a vowel before the "y", the "y" is changed to "i" and "es" is added (e.g. *he says, he tries*).

3rd person present plural: uses verb infinitive (e.g. *they have*).

3rd person past singular and plural: adds "ed" to the verb infinitive (e.g. *it killed*). If the verb ends in "y" (and there isn't a vowel immediately before the "y"), the "y" is changed to an "i" before the ending is added (e.g. *he tried, they played*). If the verb ends in "e", only "d" is added (e.g. *he loved*).

Present participle: adds "ing" to the verb infinitive (e.g. *doing*). If the verb ends in "e", the "e" is removed (e.g. *loving*).

Passive: adds "ed" to the verb infinitive - special rules as for past forms (e.g. *he was killed, it was tried*).

Negative forms: the appropriate form of "do" is chosen and used with "not" and the verb infinitive. (e.g. "does not eat" "did not say")

Nouns:

Plural: adds "s" to the singular form (e.g. *tables*). If the noun ends in "s", "x", "h" or "z", "es" is added (e.g. *foxes*). If the noun ends in "y" and there isn't a vowel immediately before the "y", the "y" is changed to an "i" and "es" is added (e.g. *boys, babies*).

There are many more rules that could be incorporated here, but it is easier to enter nouns and verbs requiring those rules as exceptions (using **add-lex**). These rules are just intended to be a quick and easy way to cut down on the number of entries required in the lexicon.

6.2 Details of English Response Generation

The system has a natural language generation facility which can give a more helpful and fuller answer to a question than just a simple yes or no, or can be used to "say" input facts or groups of facts.

The response generator takes a list of wff names, and tries to combine the information in them in a way that gives natural sounding English output. Each wff is used to create one or more fragments which are then combined and filled in with determiners, etc. If necessary, information is retrieved from the knowledge base to complete a sentence. This section describes some of the heuristic rules and assumptions used in generating English.

6.2.1 The Grammar

The English generated conforms to the mini-grammar specified in Figure xxx.

[] *indicates optionality,* * *indicates 0 or more occurrences,* | *indicates alternatives*

```
S -> | [ADVS] NP VP [ADVP]*   |
     | [pre] S, S, ... conj S |

NP -> | [determiner] [AP] noun [PP]* [Relclause]*  |
      | pronoun                                    |
      | Proper-Noun                                |
      | [pre] NP, NP, ... conj NP                  |
      | "that" S                                   |
Relclause -> \{"who","that"\} VP
AP -> | [adv]* [adjectives]*      |
      | [pre] AP, AP, ... conj AP |
PP -> [pre]* prep NP
```

```
VP -> | [adv]* | verb  [particle]* [object]* |
      | be [adv]* [object]*                  |
      | [pre] VP, VP, ... conj VP            |
object -> | NP |
          | PP |
          | AP |
          | VP |   (with aspect infinitive, passive or prespart)

ADVP -> | [adv]* prep S                    |
        | [pre] ADVP, ADVP, ... conj ADVP |

ADVS -> | [adv]*        |
        | [pre] prep NP |

conj -> and | or | but | ...
pre  -> both | either | if | ...
adv -> usually | only | ...
```

Response Generator Mini-Grammar

6.2.2 Stages Involved in Response Generation

There are several stages in the generation of English from logical form:

1) **Filtration** – removal of known or obvious clauses

2) **Organization** – organizes clauses

3) **Verbalization** – say the logical forms in English

   a. Create and combine fragments for literals
      using predicate translation information

   b. Move negations to most natural place, copy tense, passivize if necessary.

   c. Fill in gaps in sentence fragments (determiners, etc)

   d. Determine correct lexical entries for fragment contents using information from the lexicon

   e. Postprocess & Print – change "a" to "an", combine words, print

Note that the old response generator (for **ECoNet**) also had an assembly stage to change the clause form back into logical form - this is no longer necessary as the normalization **EPILOG** does is minimal. The following sections elaborate on the actions and heuristics involved in each stage of response generation.

6.2.2.1 Filtration

When responding to a question, we don't necessarily want every detail that was used in answering the question to be printed. Obvious facts should be left out, unless specifically requested by the user. The

system attaches a numerical value to each wff called its "likelihood of being known", and uses this to help sort out what information to include in an answer and what to leave out. Wffs with a high likelihood of being known can be left out of the answer - those with a low likelihood should be left in. Note that for input knowledge and story facts, we do not filter out this obvious information - it hasn't become obvious until it is said at least once!

Set-of-support clauses are automatically filtered out. Also, wffs whose complexity is greater than a threshold (***max-response-complexity***) are filtered out (because they are too costly to say). If desired, the user may be prompted for each of these to say whether it should be included or not (if ***prompt-if-too-complex*** is t - the default is nil - to just ignore those formulas). After those, the likelihood of each clause being known is compared against a threshold (***filter-threshold***). If the likelihood is higher that that threshold, and the clause is not the only clause left, it is removed.

The following heuristics are used to determine the likelihood of a clause being known:

Simple type predications (eg *[LRRH girl]*) involving constants, and specialist evaluations have the highest likelihood of being known (90) as they are the most obvious. "Sort" predications (like *(e1 episode)*) have an even higher likelihood (101).

Next, clauses involving only variables (i.e. general information in the form of rules) have likelihood of being known (80).

Clauses involving mixed constants and variables (i.e. "rules" about specific individuals, or quantification over members of a known set) have the next highest likelihood of being known (50).

And finally, clauses involving all constants (i.e. story facts other than type preciations) have likelihood of being known (40).

A future enhancement will be to also take the subjective and objective probabilities of wffs into account in calculation of likelihood.

6.2.2.2 Organization

This stage orders the wffs so that facts come first and are all in the same sentence (if possible), and each conditional is in a separate sentence after. While doing this, the organization stage tries to preserve the order given by filtration - in order of likelihood of being known, with negations kept together at the end. Episodic predications of types, and other uses of those episodes are removed here - they are not said nicely, and if necessary, the type itself can be extracted later.

6.2.2.3 Verbalization

This phase actually prepares and "says" the English response. As this is the most complex stage of the process, it has been broken down into several sub-phases. This section briefly describes what happens during each and the heuristics and assumptions involved.

6.2.2.3.1 Fragment Creation and Combination

First, a fragment representing what each wff contains is created, recursively using translation information stored with the predicate, and any operators involved in either the predicate or arguments. This

translation information is in the form of a specific "bit of grammar" to use when saying the particular construction. Numbers in this translation information represent arguments to the predicate or operator. How to set up this translation information is discussed in Chapter 3. Missing translation information may either be prompted for, or defaulted (controlled by the flag ***default-trans***).

For example, literal

[cape1 red]

contains adjective information,

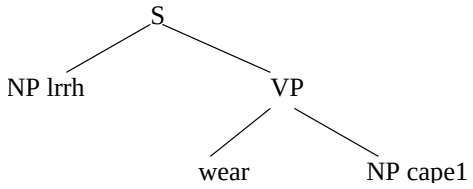
AP cape1

adjectives: RED

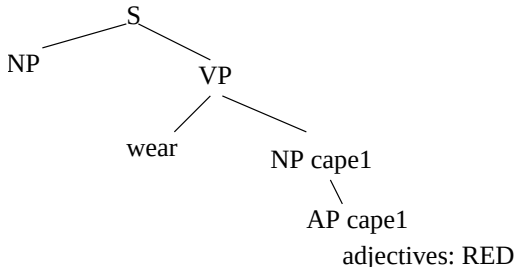
while literal

[LRRH wear cape1]

contains verb information.



The resulting fragments are combined in various ways depending on the wff they were involved in (conditionals, episodics, disjunctions, etc are all handled slightly differently). This puts noun-like information about subjects together with verb-like information about them, etc. This is done for each wff in the list, and then the fragments resulting from all of them are combined (if possible). For example, the above two wffs would be combined into a fragment for the sentence



LRRH wear red cape1

(after filling) => *Little Red Riding Hood wears a red cape*

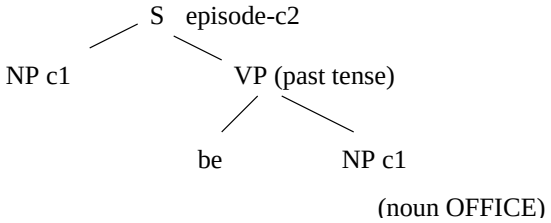
The fragments left at this stage each respresent a sentence to be output.

6.2.2.3.1.1 Episodic Wffs

All verbs are third person only, and only simple tenses are handled. The default verb tense is present. Episodic constructions yield sentence fragments, and if the episode is a constant, tense information as well - a constant episodic argument is assumed to indicate past tense. For example,

*((c1 office) ** episode-c2)*

would become a sentence fragment for

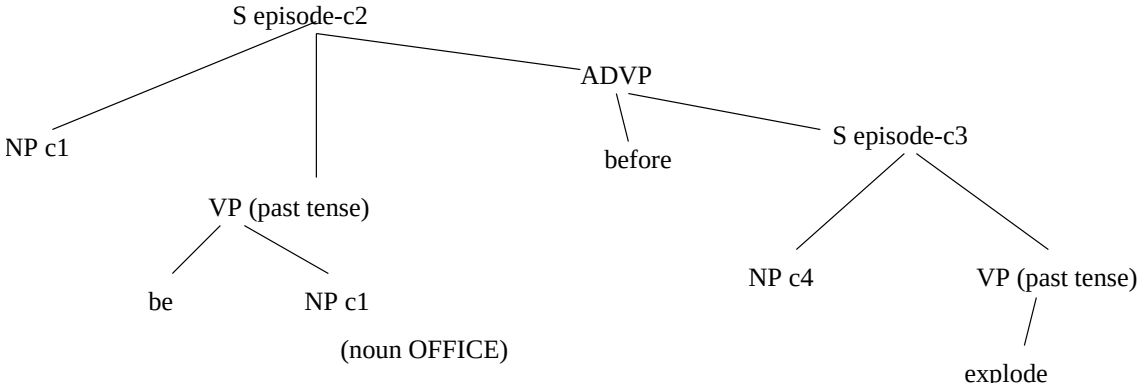


c1 was office

and would be used as a sentence, even though it contains only noun information. The name of the sentence is kept with it (the episode) so that temporal and causal adverbials can be combined with the sentence information. For example, the above wff with the following wffs,

*(episode-c2 before episode-c3), ((c4 explode) ** episode-c3)*

would combine into a fragment for the sentence



c1 was office before c4 exploded .

(after filling) => *A room was an office before a bomb exploded .*

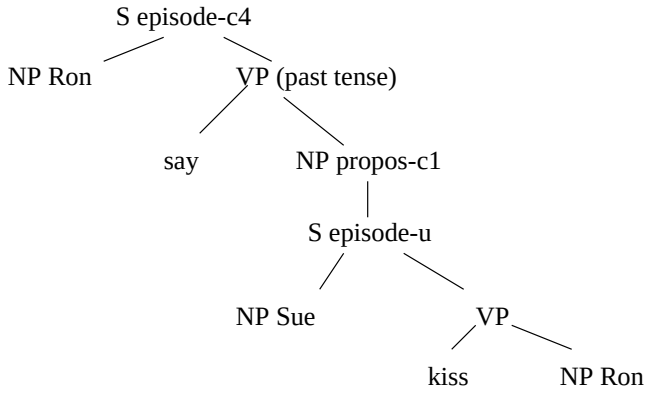
6.2.2.3.1.2 Modal Wffs

Wffs of the form *((...) _ p1)* are usually paired with other wffs like *((John say p1) * ep1)* . The propositional constants (or variables) generate an NP fragment. In the first wff, the NP will have information below it (usually an S, which gets printed as "that S"), and in the second, the NP is the object of the verb *say* . When combined, we get a fragment for *John say that ...* . For example,

*((E episode-u ((Sue kiss Ron) ** episode-u)) _ propos-c1)*

*((Ron say propos-c1) * episode-c4)*

which when combined, yield



Ron said that Sue kiss Ron

(after filling) => *Ron said that Sue kissed him*

Note that the fragments will not actually be printed (for brevity) in the next sections - those sections just describe how the fragments are combined.

6.2.2.3.1.3 Disjunctions/Conjunctions

If all literals represent the same kind of fragment (NP, VP, AP or S), a conjunction of those fragments will be created, using the appropriate conjunction (*and* , *or*). Examples:

John is a man or a boy (NP)

Human teeth are white but not sharp, big, or pointy (AP)

John ate some cake and walked to the store (VP)

John ate some cake or the wolf ate it (S)

John ate some cake and Mary ate a cookie (S)

If they are all different fragment types, the lowest common "head" type will be used (and NP for AP, PP, and NP, S otherwise). For example,

Mary is pretty or not a girl (from an AP and an NP).

6.2.2.3.1.4 Conditionals

In general, the preconditions describe the subject (NP) and the post conditions describe what happens to it (VP). (e.g. *Wolves are fierce, Everyone likes to eat people food*). This can change however, to make more natural sounding responses.

If the right hand side is a type predication, it is made the subject (if possible), and adverb "only" added in front of it. *(e.g. Only living things eat, Only living things are killed)* . If the type predication is negated, the "only" is not added, and "no" is used as the determiner (e.g. *No girls or women kill anything*).

If the left hand side (preconditions) has a verb associated with it, the verb is made into a relative clause. *(e.g. Anything that eats is alive)* .

Complicated implications (with more than one literal after combining the right hand side) are handled by saying "if preconditions, postconditions" *(e.g. If John eats something, he is hungry or it is yummy)* .

The objective probability and quantifiers are used to give an adverb or quantifier for the sentence (e.g. *Most wolves are grey* , *If a bomb blows up near a location while a physical object is in the location, the object is usually destroyed*).

6.2.2.3.1.5 Lambda Expressions

A lambda expression is converted into a VP fragment, using translation information (as if it were a quantified wff). Then the subject of the VP fragment (a lambda variable) is replaced by the first argument in the wff that had the lambda expression in it. For example:

*((E x (x creature) (E y ((x want (To (L u (L v (E z ((v eat z) ** u)))))) ** y)))*

*0.9 (E w (w cause-of y) ((x hungry) * w)))*

is said as:

Creatures that want to eat are usually hungry .

There is a problem with saying the lambda expressions if there is a reference to the lambda variable other than as the subject of the main formula within the lambda expression - the system isn't smart enough (yet) to figure out what to use in place of that lambda variable.

6.2.2.3.2 Rearranging for Negation and Passive

Negations are moved to the most natural place in the sentence. For example, the negation is moved from a verb object to the verb, and then to the subject to get

No fat things are thin

instead of

All fat things are not thin .

The negation is moved from and adverbial to the verb phrase to get

The wolf did not eat Little Red Riding Hood before it ate Grandmother

instead of

The wolf ate Little Red Riding Hood not before it ate Grandmother .

If a verb is already negated when we move a negation there, both negations are removed, and the adverb "only" inserted so that we get

All humans only like to eat people food

instead of

All humans do not like to eat not people food .

Tense is copied among fragments as well - if any of the sentence fragments involved have past tense, this is copied to related sentences as well. (Thus in the example shown earlier, we have *Ron said that Sue kissed him* even though there was no tense information on the *Sue kissed him* fragment). Also, if two sentences are in a conjunction or adverbial, and the object of one is the subject of the other, it is passivized.

Nothing is alive after it is killed

instead of

Nothing is alive after something kills it .

Also

All creatures that wear apparel are covered by it

instead of

If creatures wear apparel, the apparel covers them

Passivizing is done by creating a new fragment where the object is now the subject, the verb is now a combination of the appropriate form of the verb *be* and the passive form of the original verb, and a new verb object has been added which is a PP *by* the original subject. The verb *be* , and verbs with passive form **'no-passive** cannot be passivized. If the original subject has no information on it and is not a "required" fragment, it may be left out of the final sentence (as it was in one example above).

In some cases, we may passivize a verb phrase or have a passive form from the original translation, and it may be more natural to say that in its active form. It is fairly easy to do the transformation the other way, but this will remain as a future enhancement for now.

6.2.2.3.3 Filling Gaps and Building Descriptions

Now any gaps in the fragments in the final list of fragments are filled in, including number and determiner for noun phrases, and descriptions for entities if necessary. These steps have some complicated heuristics and so are discussed in separate sections shortly. In addition, there are a few other "rules of

thumb" used.

To "say" a quoted expression, a specialized routine associated with the sort of the quoted expression is used (so far sets, times and numbers are handled). Sets are expressed as a complex noun phrase using *and* or *or (e.g. Grandmother, Little Red Riding Hood, and the man ...)* , absolute times are said as a time on a date *(e.g. 12:00 on March 10, 1988; some time on some unspecified day in June, 1987)* , and numbers are just printed as themselves *(e.g. 35)* .

If no verb information is present for a sentence, the fragment itself is split into a subject and object, and either the verb *have* (for part-of relations – for example, a noun phrase representing *tail of wolf* would become *wolf has a tail*), or *be* (all other cases – for example, *LRRH girl* would become *Little Red Riding Hood is a girl*) is used. Agreement of subject and object number for "be" verbs is done here as well.

If a noun phrase for a variable has noun *member* and a PP *of* with an NPconj beneath it, it usually indicates that every one of the members of that set did something, so only the set members need be mentioned (*Grandmother, Little Red Riding Hood, and the man talked* rather than *Every member of Grandmother, Little Red Riding Hood and the man talked*).

6.2.2.3.3.1 Retrieval of Information

If an entity has been referenced in one of the fragments, but so far no noun-like or sentence-like information describing the entity has been found, the system will look in the knowledge base to find the missing information. This will be made into a fragment and combined with the other fragments to say. For episodes, it looks under *(item episode-specialist)* , for propositional arguments under *(item _ object)* , and for physical objects, under *(item type-specialist)* (for the type). These new wffs are made into fragments just as described earlier, and attached to the fragment in question. In the *lrrh wear cape1* example earlier, the type of *cape1* was looked up this way (also for *c1* and *c4* in the episodic example). Saying single wffs like *(episode-c1 before episode-c2)* would cause *(episode-c1 episode-specialist)* and *(episode-c2 episodie-specialist)* to be looked up. Saying *((john say p1) * episode-c1)* would cause *(p1 _ object)* to be looked up.

6.2.2.3.3.2 Selection of Number

Selection of number is only necessary for variables (constants are singular for now unless an operator indicates otherwise). If the noun is "person" or "thing", singular is used wherever it makes sense (*everything* sounds better than *all things*). Otherwise plural is used for the first variable. *(e.g. All wolves like to eat creatures*). For variables after the first, the same number as the first variable is used *(e.g. Everyone loves something)* . If a negation is involved, the other number is used (plural instead of singular and vice versa) *(e.g. Nobody makes inanimate natural objects, The wolf is not smaller than any basket)* .

6.2.2.3.3.3 Selection of Determiner

For constants, a skolem constant gets "a", a non-skolem one gets "the" in subject position, "a" in object position. A mass noun gets determiner "some". *(e.g. Little Red Riding Hood wore a red cape, The wolf is a wolf, Some honey was in the basket)* .

For variables, negated items get determiner "no" *(e.g. No fat things are thin).* If a negation was used earlier in the sentence, and the current item occurs in a positive context, "all" or "every" will be

used (plural and singular, respectively). *(e.g. Nobody likes everyone)* . If the item occurs in a negative context, the determiners are "any" (plural or for nouns *thing* or *person*), or "a" (singular). *(e.g. The wolf is not smaller than any baskets, No human has a tail)* Otherwise, the first variable in a sentence gets a determiner based on the context it occurs in and quantifier information, and whether or not the noun is a mass noun. For example, for quantifier *A* , the generator uses "all" (plural or mass noun) or "every" (singular). *(e.g. Every wolf has a tail, Most wolves are grey, Little Red Riding Hood likes to eat all cakes*). Otherwise, plurals and mass nouns get no determiner *(e.g. All wolves like to eat creatures)* , while singular objects get determiner "a", unless they have noun "person" or "thing" in which case they get determiner "some". *(e.g. Everyone likes something)* For singular nouns which have a relative clause attached, "any" is used instead of "every". *(e.g. Anything that lives in something is smaller than it*).

If there are more than one identical noun phrases, the second one will get "another" as its determiner (or "other" if it is plural), and will be referred to as "the other" (subject) or "that" (object) for referring noun phrases. If there is a third, it gets the determiner "yet another" or "even more" (plural) and is referred to as "the last" (subject) or "that last" (object) ("those last" for plural). *e.g. Anyone that is told something by another person usually believes it.*

If a noun phrase has a PP *of* below it, and the noun is not *part* , an attempt is made to make the noun phrase into the possessive *noun2's noun* instead of *noun of noun2* , with *noun's* being added as the determiner of *noun2* , and the PP removed. This was used in the *cardinality of* example earlier. Also in *There is a woman that is Little Red Riding Hood's mother* .

6.2.2.3.3.4 Referring Noun Phrases

If an entity has been used already in the current sentence (or a previous sentence in the group of sentences being printed), a pronoun or shorter phrase is attempted instead of the whole description. Pronouns are selected based on the number of the entity, the noun used in the original description, and whether the current entity is in subject or object position. Names may have pronouns associated with them as lexical information (for example, *lrrh* has "she" for a subject pronoun, "her" for an object pronoun). Plural entities and entities whose noun have with property *group* (i.e. *group, set, collection*) use pronouns "they" or "them". Nouns with property *masculine* (i.e. *person, man, boy, human, father* and *terrorist*) indicate "he" or "him". Nouns with property *feminine* (i.e. *woman* and *girl*) indicate "she" or "her". Everything else will use pronoun "it".

Some examples:

John said that Sue kissed him.

Ron believed it.

Anyone that says something usually believes it .

Little Red Riding Hood met a wolf.

It was near her.

She saw it.

If the use of a pronoun is ambiguous (i.e. there are other entities that would generate the same pronoun), a short phrase is used, which consists of the noun of the original fragment, and adjectives as well if the noun alone is ambiguous, and determiner "the".

If a bomb blows up outside a location while a physical object is in the location, the object is usually destroyed

If there are several entities with identical descriptions, and the system has added "another" or "other" as a determiner for one, any referring phrase for those entities will use "the other" or "that".

If an event is another event's cause, the other event is after it.

Anyone that kisses another person usually loves that person.

Possessive pronouns are handled when a noun phrase *noun of noun2* is converted to *noun2's noun* (as described earlier). If *noun2* is an entity which has been mentioned before, the appropriate pronoun is decided for it, and then the possessive form determined ("his", "her", "its", "their").

When either children's fathers or their mothers live in buildings, the children usually live in the buildings.

The system also attempts to use reflexive pronouns where appropriate. So far the only rule it uses is that if the direct object of a verb is the same as the subject, a reflexive should be used. Thus we can get

The wolf likes to eat itself .

However, there are other cases where reflexives should be used that the response generator cannot handle yet. For example,

Creatures that want to eat food for them are usually hungry

should use *themselves* instead of *them* .

6.2.2.3.4 Determination of Lexical Items

The major work has now been done, and the next step is to go through each fragment and determine the lexical item to use for each part. It looks this up in the lexicon, which contains the different forms of verbs and nouns, and the strings corresponding to adjectives and other items. Input to the lexicon is described in Chapter 3. If lexical information is missing, the system will either prompt the user for it, or build the item using its default rules (controlled by the flag ***default-lex*).**

6.2.2.3.5 Postprocess for Word Combinations and Print

A postprocessor then goes through the output comparing all consecutive pairs of words, changes "a" to "an" before a vowel, and combines words that make up a compound into one word. The combinations are:

every thing/person => everything/everyone

no thing/person => nothing/nobody

any thing/person => anything/anyone

some thing/person => something/someone

Human is treated exactly the same as *person* during the combination.

Then the sentence(s) is(are) printed. If this is an answer to a question, the first sentence contains the answer followed by a comma (Yes, ... or No, ...).

6.2.3 Problems

The latest version of the response generator uses a somewhat more general approach than its predecessor, but it is still rather ad-hoc, using a few heuristics which are good enough most of the time, but may easily prove inadequate with additional test data. Please keep this in mind and if you notice "strange" sounding output, report it to the authors so that the next version can be improved.

Currently objective probabilities of rules are taken into account (using quantifiers like *most* or adverbs like *usually*), but subjective probabilities are not. This may or may not be desirable, but should be investigated anyway.

Although the system can now handle various operators and complex syntax now, it is still quite a trickly process to determine the exact translation to give it, and there are some problems with combinations.

For question answers, the current filtering mechanism does not take the question itself into account. There are some instances where more clauses should be filtered out (e.g. in answer to the question *Is every rock a human?* , the answer is *No, there is a rock*). In such cases, we really need to filter out the presuppositions of the question.

Glossary of Terms [1]

axiom schema - a higher level rule which creates an axiom for each component which matches the schema - this is like quantifying over predicates, variables, operators, formulas, etc. (as used in **EPILOG)**

backchaining - the recursive process of using rules in their reverse direction to create new subgoals in an attempt to prove a formula.

conditional - a rule; that is, a formula which has an antecedent, which when matched, results in a consequent being generated.

contrapositive - rules state that when this antecedent is satisfied, the consequent follows, but in contrapositive reasoning, the rule is applied differently, saying that if the consequent is false, the antecedent is too (otherwise we could infer the consequent, which we already know to be false).

episodic logic - a first order logic with extensions to specifically handly episodic constructions (those formulas which need events associated with them for temporal and causal connections).

forward inference - the recursive process of matching antecendents of rules, and generating their consequents, which may also match antecedents ...

generalized resolution - a more general resolution inference rule where strict incompatibility is not required, and the predicates and arguments do not have to be identical (for example, (t1 before t2) and (t2 strictly-before t1) are resolvable using generalized resolving, but not by traditional resolving).

goal-driven inference - the recursive process of applying rules to a goal or subgaol in an attempt to solve it, and generating subgoals from this which may then lead to a solution. This is usually done in the backward direction (backchaining), but may also be done in the forwad direction (contrapositive inference).

goal reduction - another name for goal-driven inference - reducing a goal to smaller subgoals

input-driven inference - the recursive process of applying rules to input or inferred facts, and generating consequents from these. While this is most commonly done in the forward direction (forward inference), it may also be done in the reverse direction (contrapositive reasoning).

input chaining - another term for input-driven inference.

meaning postulate - usually a rule which is required to divulge the meaning of a word (such as *maim -> wound*). In **EPILOG** the term is used often to speak of axiom schemas for classes of words.

natural deduction - an inference mechanism based on a number of inference rules, which apply to specific connectives. There are typically two rules of inference for each connective - one that introduces it into expressions and one that eliminates it.

[1] With a little help from: Avron Barr and Edward Feigenbaum, **The Handbook of Artificial Intelligence** , Volume 1, William Kaufmann, Inc, 1981

nominalization - making a noun or noun phrase out of a verb or sentence.

presupposition - assumed to be true when a statement or question is uttered. For example, the question "Has Joe stopped beating his wife?" has that "Joe beats his wife" as a presupposition.

resolution - an inference mechanism based on a single inference rule which looks for incompatibility of literals between a questioned formula and known facts and rules.

skolemize - replacing an existentially quantified variable in an assertion with a new constant - the constant is called a *skolem* constant.

semantic net - a representation mechanism consisting of nodes (for objects, events, concepts) and links between the nodes specifying their relationships.

wff - well formed formula - that is, a syntactically correct logical formula.

References and Further Reading

Episodic Logic:

Hwang, C.H., **A Logical Approach to Narrative Understanding** , PhD Thesis, University of Alberta, 1992.

Schubert, L.K., and Hwang, C.H., *A Logical Approach to Goal-Based and Causal Inference in Narrative Understanding* , BCS-G2010-72, The Boeing Company, 1988

Schubert, L.K., and Hwang, C.H., *Linguistic and World Knowledge in Story Understanding: A Logical Approach* , prepared for Boeing under purchase contract W-278258, 1988.

Schubert, L.K., and Hwang, C.H., *An Episodic Knowledge Representation for Narrative Texts* , **Proceedings of the First International Conference on Principles of Knowledge Representation and Reasoning (KR'89)** , 1989, pp. 444-458

Schubert, L.K., and Hwang, C.H., *An Episodic Knowledge Representation for Narrative Texts* , TR 345, University of Rochester, Rochester, N.Y., May 1990.

Specialists and ECoNet:

Schubert, L.K., Papalaskaris, M.A., Taugher, J., *Accelerating Deductive Inference: Special Methods for Taxonomies, Colours, and Times* , **The Knowledge Frontier** , N.Cercone and G. .CCalla (ed), 1987

deHaan, J., Miller, S., and Schubert, L.K., *The User's Guide to ECoNet* , prepared for Boeing under purchase contract W-278258, 1988.

Miller, Stephanie, and Schubert, Lenhart K., *Using Specialists to Accelerate General Reasoning* , **AAAI-88** , St.Paul, Minnesota, 1988, pp. 161-165

Miller, Stephanie, and Schubert, Lenhart K., *Time Revisited* , **Computational Intelligence** , Vol. 6, No. 2, May 1990, pp. 108-118

Kaplan, Aaron, *A Computational Model of Belief* , PhD Thesis, University of Rochester 2000.

Index