EPILOG: The Computational System for Episodic Logic

LIBRARY ROUTINES

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

Introduction

This manual describes the calling sequence and actions of some of the low level routines used by **EPILOG** . It is intended for the user building his/her own specialist, or for a system maintenance person trying to track down a bug.

There are several sets of routines described here: the Chapter 2 contains routines for symbol manipulation, tweaking parameters, retracting information, etc, all of which are potentially useful for any system and are not **EPILOG** dependent. Chapter 3 contains the low level routines which deal with specific **EPILOG** internal structures. Chapter 4 contains routines needed to interface specialists with **EPILOG** and with other specialists.

Chapter 2

General Routines

This chapter contains the calling sequences and actions for a number of routines for symbol manipulation, tracing system actions, tweaking parameters, help facilities, hashing routines, output routines, and checkpointing and retraction facilities. The routines is this section are useful for any large system and are not **EPILOG** dependent. They are located in directory *epi/lib/* , and a

(require 'lib "..EPILOG path ../epi/lib/lib")

command will load them into your workspace, and you can do a *use-package* or use whatever access method you prefer to use them.

2.1 General Routines

2.1.1 Symbol Manipulation

(pack *symbol-list*) [*function*]

This function squishes all of the symbol names in the list *symbol-list* into a new symbol name. For example, *(pack '(a b c))* would give a new symbol *abc* .

(pack* *&rest symbol-list*) [*function*]

This function squishes all of the symbol names in the list *symbol-list* into a new symbol name (just another interface to the **pack** routine.

(unpack *symbol*) [*function*]

This function explodes *symbol* into a list of the characters that make up its name. If *symbol* exists in a package other than the one this routine is called from, the package name will also be exploded and at the front of the list.

(strip-pkg *arg*) [*function*]

returns the exploded version of *arg* (a list of characters) with the package name removed from the front.

(build-symbol *symbol-list*) [*function*]

builds a new symbol from the symbols in *symbol-list* . Exactly like **pack** except that package names are stripped off first.

(character-atom *code*) [*function*]

returns the character responding to *code* .

(nthchar *n symbol*) [*function*]

returns the *n* th character in the list of characters making up *symbol* 's name.

2.1.2 List Manipulation

(filter1 *pred list*) [*function*]

returns a list consisting of the first element of *list* which satisfies *pred* , followed by the remaining elements of *list* . If none satisfy *pred* , the result is nil.

(filter *pred list*) [*function*]

returns all the elements in *list* that satisfy *pred* .

(revers *x*) [*function*]

calls the lisp function **reverse** iwth *x* after checking to see if it is a list or an atom.

(last1 *x*) [*macro*]

returns the last ELEMENT of *x* (the CAR after using the function **last**).

(nondest-rplacd *old item new*) [*function*]

non-destructively replaces the thing associated with *item* in *old* with *new* . *Old* is assumed to be an association list.

2.1.3 Other

(average *&rest items*) [*function*]

computes the average of all the numbers in *items* . *Items* must all be numbers.

(squish *strings*) [*function*]

concatenates all the strings in the list *strings* together into a longer string.

(check-online-status) [*function*]

returns t if the current process is running interactively, nil otherwise. This is useful to check if a program does any querying of the user.

2.2 Output Routines

2.2.1 Warnings and Errors

(print-error *&rest print-stuff*) [*function*]

prints a message beginning with "ERROR: ", followed by *print-stuff* .

(fatal-error *where &rest print-stuff*) [*function*]

prints a message beginning with "ERROR: ", followed by *print-stuff* , and then jumps to *where* . The lisp command **throw** is used, so *where* had better be in a **catch** .

(warn-error *&rest print-stuff*) [*function*]

if *warn* is t, prints a message beginning with "WARNING: ", followed by *print-stuff* .

warn [tweakable]

if set to t, warnings will be printed. Otherwise they are suppressed. The default is to print warnings (t).

2.2.2 Printing

(print-line *&rest print-args*) [*function*]

This function can be used by a specialist display routine to print *print-args* onto standard output after starting a new line and spacing the current indent spaces. If ***nl*** is a member of that list, a new line will be printed in its place.

(print-it *&rest print-args*) [*function*]

This function can be used by a specialist display routine to print *print-args* onto standard output without starting a new line.

nl [constant]

when included in a list of items to be printed will cause a new line to be started.

print [tweakable]

global "allow printing" flag. If set to nil, these print routines will not print.

2.2.3 Indenting

(set-indent *number*) [*function*]

This function sets the current indent to *number* , and sets *spaces* to the appropriate number of spaces. Whenever **print-line** is called, the *spaces* space string will be printed first - automatic intenting.

(reset-indent *spaces number*) [*function*]

This function resets the current indent to *number* , and the current space string to *spaces* .

(indent *number*) [*function*]

starts a new line and prints *number* spaces.

(depth-indent *number*) [*function*]

starts a new line and prints *number* * 2 spaces.

spaces [variable]

string containing spaces which is printed at the beginning of each **print-line** command. This should be set using the **set-indent** and **reset-indent** commands.

indent [variable]

contains the number of spaces which will be printed at the beginning of each line. This should be set using the **set-indent** and **reset-indent** commands.

2.3 Help

help-path [tweakable]

path used to find help files. For example, for the **EPILOG** system, it is "EPILOG/epi/help/.hlp", which indicates that the help files are found in directory EPILOG/epi/help, and the files end in .hlp.

(help *&rest items*) [*function*]

retrieves information from the help files corresponding to *items* and displays it.

2.4 Hashing Routines

Hash tables are useful objects to have in a system. These routines set up expandable hash tables which use *sxhash* to calculate hash position, which allows lists to be safely used as hash keys. The tables are automatically expanded when they get too full.

(initialize-element-table *name*) [*function*]

creates an expandable hash table with name *name* . The hash table itself is returned.

(put-element *element name table &optional temp*) [*function*]

puts *name* into hash table *table* , using *element* as the hash key. If *table* is too full, it is expanded first, automatically.

Put-element assumes that the element is not there already. If *temp* is specified, a simple insert is done, otherwise the system uses the checkpointing-retraction insertion routines.

(get-element *element table*) [*function*]

retrieves the item in *table* corresponding to the hash key *element* .

element-table [structure]

this is the structure the hash table routines use. For most operations the details of this structure are not necessary, but for those where it is, the fields are as follows:

element-table-load	current number of entries stored
element-table-size	current size of table
element-table-vector	actual hash table
element-table-size-index	index used to get current size
element-table-rehash-threshold	load fraction before rehashing
element-table-current-rehash	element number that will cause rehash

2.5 Tracing

The tracing feature allows you to set up your system so that as much or as little output as is desirable at a given time, or by a given user is produced. This is handy for debugging, or intense scrutiny of system processes. To do this, you set up names corresponding to the different sets of output (using **traceable**), and test to see if they are being traced (using *traced-p*) before printing. If a number of packages are in use, it is wise to make the trace symbols internal to the package the user will most likely be in (for **EPILOG** this will be the package *epilog*).

(traceable *trace-symbol documentation tracelist*) [*function*]

Sets up *trace-symbol* as a new item which may be traced. *Documentation* is added as a property of *trace-symbol* , and is printed when **print-traceables** is used. If *tracelist* is specified, *trace-symbol* is a complex trace item which will turn tracing on or off for all of the symbols in *tracelist* when tracing for it is turned on or off. Tracing is turned on and off using the **trace** and **untrace** functions. Note - all user functions handled by the specialist should test to see if *fn-eval* is being traced, and if so, print out the function they compute and its result.

(traced-p *symbol*) [*function*]

returns t if *symbol* is currently being traced, nil otherwise.

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

starts tracing for all entries in *items* . Each is checked to ensure it is a traceable item first, and an error message printed if it isn't.

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

stops tracing for all entries in *items* .

(trace-all) [*function*]

starts tracing for all traceable items.

(untrace-all) [*function*]

stops tracing for all traceable items.

(print-traceables *&rest items*) [*function*]

prints traceable items, with their documentation. If *items* are specified, they are printed only if they are legal trace values. If no items are specfied, all traceable values will be printed. Currently traced values are also printed.

traceable-items [variable]

contains all the items which may be traced.

traced-items [variable]

contains the items currently being traced.

2.6 Parameter Tweaking

In a large system, there are often a number of system parameters which the user can set to make the system operate differently. These are sometimes difficult for both user and system designer to keep track of, and often a user will change a parameter and not get the desired result, and be unable to reset it to the original value. To get around this, the tweak feature has been set up. The **tweak** command is very much like a **setq** , except that if no new value is given, a default may be obtained and used. Documentation is attached to each parameter as well.

(tweakable *parameter &optional default documentation type-spec*) [*function*]

Sets up a tweakable parameter *parameter* with initial value *default* . This may be then changed using the **tweak** command. *Documentation* is added as a property of *parameter* , and is printed when **print-tweakables** is used. A LISP type specification can also be added to constrain possible values for the parameter (e.g. *integer* , *(integer 1 100)* , *boolean* (t or nil), etc).

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

sets the value to *item* to *value* . If *value* is not specified, the default from *item* 's property list is used.

(print-tweakables *&rest items*) [*function*]

prints tweakable parameters and their documentation. If *items* are specified, they will be printed only if they are valid tweakable parameters. If no items are specified, all tweakable paramters will be printed.

tweakable-items [variable]

contains all the parameters which may be tweaked.

2.7 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 for this, the display commands were set up. These allow you to set up a printing function, and an easy to remember name for it, and then the user need only call **display** with that name.

(set-display-function *name function-name &optional documentation*) [*function*]

Extends the **display** command, by adding the symbol *function-name* as a valid display option. Whenever **display** is called with this symbol as the display option, it will invoke the function *function-name* , which should be a function with the parameter list *brief-flag* **&rest** *display-args* .

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

args is a list consisting of a display option, possibly arguments to it, and optionally ONE of the flags *-brief, -b* , *-full,* or *-f* . The flags are stripped off, and the routine corresponding to the display option will be called with the rest of the arguments, as well as a "brief" flag. **Display** with no arguments will cause the display options to be printed. A '-key ('-k) flag is accepted for some display options (*tweak* , *trace* , and *display*). In this case apropos is used on each argument, and the valid options 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.

(print-display *&rest items*) [*function*]

prints display options and their documentation. If *items* are specified, they will be printed only if they are valid display options. If no items are specified, all display options will be printed.

display-items [variable]

contains a list of all the options which may be used in the display commmand.

display-default [tweakable]

contains the name of an option to use if there is no option specified by a display request, and the arguments are not know tweakable parameters or trace values. That function is then invoked with the arguments to the display command.

2.8 Storage Functions

This section describes routines for storage at a low level, including routines whose actions may be later retracted.

2.8.1 Checkpointing and Retraction

These routines allow you to set checkpoints during a session and to retract anything entered after that checkpoint. This does not include changes to system parameters or the list of items being currently traced. Checkpointing and retraction are virtually identical to the same functions in **ECoNet** . 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 have the application define arbitrary "units" (for example, **EPILOG** uses formulas), and to allow retraction of a specified number of these units. The two methods may be mixed so that you can retract a number of objects while still keeping track of a named checkpoint.

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

This function should be called before a specialist modifies its knowledge base. It adds a symbol to the checkpoint rewind stack, to mark a place that the user may rewind back to. 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. **checkpoint** does nothing if checkpointing is not currently enabled; **checkpointing-p** can be used to test whether a checkpoint has actually been set.

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

where item is a number (the number of anonymous checkpoints to retract), or a symbol (the checkpoint to retract to). In the latter case, if a keyword argument *:remove t* is also given, the checkpoint itself will be removed after the retraction. **(display 'checkpoints)** can be used to see what checkpoints have been named.

(checkpointing-p) [*function*]

Returns true if checkpointing is currently enabled. This prevents doing a lot of unnecessary work to save something.

(get-last-checkpoint) [*function*]

Returns the last checkpoint set. The only valid use of the value returned by this function is to determine if a checkpoint has been set since a previous invocation of **get-last-checkpoint** , by **eq** comparing the two values.

(start-checkpoint *undo-function undo-args*) [*function*]

This function will put an item on the rewind stack. When a retraction is called for, *undo-function* will be called with *undo-args* . This should be used when enough atomic level changes are grouped together that it is wasteful to have a separate item for each. For example, wffs in **EPILOG** have their own rewind routine called **remove-wff** , which just sets the property list of the wff symbol to nil. This avoids having separate rewind items for each property of the wff.

(anon-checkpoint) [*function*]

this starts up one of the "numbered" variety of checkpoints. Retraction using numbers will retract to checkpoints created with this function. For example, in **EPILOG** , each input routine for formulas contains a call to **anon-checkpoint** so that retraction of a specified number of formulas can be achieved.

(display-ckpts *&optional msg print-stack*) [*function*]

prints the currently active checkpoints, optionally preceded by *msg* . If *print-stack* is t, the rewind stack itself is also printed (not recommended except for debugging! It's very long!!)

additional-checkpoint-routine [tweakable]

contains the name of an additional routine to call whenever a checkpoint is started. For example, if you are maintaining a global environment of some sort and wish it to be automatically saved and restored during checkpointing and retraction, you could set this to the name of a routine which added another item to the rewind stack for the environment.

checkpoint-names [variable]

contains the names of the currently active named checkpoints. Order is not guaranteed.

2.8.2 Storage for Retraction

These routines automatically add information to the rewind stack so that the changes may be retracted later.

(change-array *array-name index newvalue*) [*function*]

This function will change the array entry for *index* in *array-name* to *newvalue* , after saving the oldvalue and adding an element to the rewind list for retraction later.

(change-hash *item hash-array newvalue*) [*function*]

This function will change the hash entry for *item* in *hash-array* to *newvalue* , after saving the oldvalue and adding an element to the rewind list for retraction later.

(change-field *field-name item newvalue*) [*function*]

This function will change the field *field-name* in *item* to *newvalue* , after saving the oldvalue and adding an element to the rewind list for retraction later.

(change-property *item property-name newvalue*) [*function*]

This function will change the property *field-name* in *item* to *newvalue* , after saving the oldvalue and adding an element to the rewind list for retraction later.

2.8.3 Storage/Retrieval without Retraction

(geta *array index*) [*macro*]

returns the element in *array* at location *index* .

(geth *hasharray index*) [*macro*]

returns the element in *hasharray* at hash location *index* .

(getp *item property*) [*macro*]

returns the contents of the property field *property* for *item* .

Note: the following routines set properties, and array locations, but do NOT add the changes to the rewind list. Use these only for things which should not change even if a retraction is done, or for things which are being "rewound" at a higher level and do not need the atomic changes saved.

(seta *array index newvalue*) [*macro*]

sets the element in *array* at location *index* to *newvalue* .

(seth *hasharray index newvalue*) [*macro*]

sets the element in *hasharray* at hash location *index* to *newvalue* .

(setp *item property newvalue*) [*macro*]

sets the contents of the property field *property* for *item* to *newvalue* .

Chapter 3

Low Level EPILOG Routines

This chapter contains the calling sequences and actions for a number of low level routines for examining, manipulating and creating **EPILOG** objects. These are all located in the *epilib* library, but a special interface called *spec* has been set up to minimize the number of conflicts due to imported symbols. The routines are located in directory *epi/epilib/* , and a

(require 'epilib "..EPILOG path ../epi/epilib/epilib")

command will load them into your workspace.

If you are building a specialist, you will also need the specialist interface routines in the next chapter. To make things easy, you can simply load them in, and you will automatically get these routines as well. I.e.

(require 'spec "..EPILOG path ../epi/spec/spec")

It would be wise to do a *use-package* very early in the specialist, or simply to add *spec* to the package use-list when defining the package. For example:

(in-package 'time-specialist :nicknames '(time) :use-list '(lisp user spec lib))

Note: these routines are intended to be operation when all of **EPILOG** is loaded - they do expect the **EPILOG** package to be present, and so cannot be simply loaded independent of **EPILOG** (unlike the routines in the previous chapter).

3.1 Information Routines

These routines return parts of **EPILOG** objects and properties about them. The list form of a normalized formula is available under the *'print* property.

3.1.1 Properties of Normalized Objects

(print-info *item*) [*function*]

returns the *print* property of an **EPILOG** object *item* , or just *item* if there is nothing under that property.

(type-of-concept *arg*) [*function*]

returns the type of the concept - will be one of *constant, variable, function* .

(term-type *term*) [*function*]

returns the type of term *term* is (*modified, constant, variable, wff, quoted-expression, quasi-quoted-expression, record, function*)

(pred-type *pred*) [*function*]

returns the type of predicate *pred* is (*modified, constant, variable, lambda*)

(op-type *op*) [*function*]

returns the type of operator *op* is (*modified, constant, variable*)

(wff-type *wff*) [*function*]

returns the type of formula *wff* is (*quantified, logical, causal, episodic, prefix, modified, constant, variable, quasi-quoted-expression*)

(entity-type *entity*) [*function*]

returns the type of entity *entity* is (*quantifier, wff, term, pred, op*)

(neg-p *wff*) [*function*]

returns t if *wff* is negated, nil otherwise.

(wff-p *object*) [*function*]

returns t if *object* is a normalized formula, nil otherwise.

(term-p *object*) [*function*]

returns t if *object* is a normalized term, nil otherwise.

(pred-p *object*) [*function*]

returns t if *object* is a normalized predicate, nil otherwise.

(operator-p *object*) [*function*]
returns t if *object* is a normalized operator, nil otherwise.

(quantifier-p *object*) [*function*]
returns t if *object* is a normalized quantifier, nil otherwise.

(function-p *object*) [*function*]
returns t if *object* is a function symbol, nil otherwise.

(epi-variable-p *object*) [*function*]
returns t if *object* is a variable, nil otherwise.

(function-term-p *object*) [*function*]
returns t if *object* is a functional term, nil otherwise.

(record-p *object*) [*function*]
returns t if *object* is a record term, nil otherwise.

(quoted-expression-p *object*) [*function*]
returns t if *object* is a quoted-expression term, nil otherwise.

(quasi-quoted-expression-p *object*) [*function*]
returns t if *object* is a quasi-quoted-expression term, nil otherwise.

(lambda-pred-p *object*) [*function*]
returns t if *object* is a normalized lambda predicate, nil otherwise.

(type-pred-p *object*) [*function*]

returns t if *object* is a type predicate, nil otherwise.

(skolem-constant-p *object*) [*function*]

returns t if *object* is a constant resulting from skolemization, nil otherwise.

3.1.2 Other Information from Normalized Objects

(find-sort *arg*) [*function*]

returns the sort of *arg* . If a concept, the sort is returned. If a record, the second element of the list.

(instance-type *item*) [*function*]

returns a list of the types which have been asserted for *item* .

(content *item*) [*function*]

returns a list of the formulas which have been asserted with _ for *item* . The formulas are all in list form.

(record-contents *item*) [*function*]

returns the part of the record without the record indicator or sort.

(description *item*) [*function*]

returns a list of the formulas which have been asserted with ****, *** , or @ for *item* . The formulas are all in list form.

(non-constant-arg *arglist*) [*function*]

returns t if any of the arguments in *arglist*is not a constant concept or a quoted expression, nil if they are all constants. Note that variables or functions with any kind of arguments are not considered constants by this routine.

(check-constant *arg*) [*function*]

returns t if the given argument is a constant - either a record or a constant concept; otherwise nil.

(number-arg *arg*) [*function*]

returns t if *arg* is either a concept of sort **number** , or a number.

3.1.3 For Unnormalized Objects (Lists)

(prop-pred *prop*) [*function*]

returns the predicate of *prop* . If *prop* is a quantified proposition, the predicate is *nil* .

(prop-args *prop*) [*function*]

returns the list of arguments of *prop* . If *prop* is a quantified proposition, this list will be *nil* .

(is-negated *prop*) [*function*]

returns t if *prop* is negated (i.e. a list beginning with *not*), nil otherwise.

(is-wff *item*) [*function*]

returns t if *item* is an unnormalized proposition, nil otherwise.

(is-pred *item*) [*function*]

returns t if *item* is an unnormalized predicate, nil otherwise.

(is-function *item*) [*function*]

returns t if *item* is a function, nil otherwise. (Note: currently functions may only be simple symbols, so **function-p** and **is-function** are identical).

(is-variable *item*) [*function*]

returns t if *item* can legally be a variable.

(is-function-term *item*) [*function*]

returns t if *item* is an unnormalized functional term, nil otherwise.

(is-quoted-expression *item*) [*function*]

returns t if *item* is an unnormalized quoted expression, nil otherwise.

(is-quasi-quoted-expression *item*) [*function*]

returns t if *item* is an unnormalized quasi-quoted expression, nil otherwise.

(is-record *item*) [*function*]

returns t if *item* is an unnormalized record term, nil otherwise.

(is-operator *item &optional result*) [*function*]

returns t if *item* is an unnormalized operator, nil otherwise. If *result* is specified, the result of the operator must be of that type, otherwise any type will do.

(is-quantifier *item*) [*function*]

returns t if *item* is an unnormalized quantifier, nil otherwise.

3.2 Creation of EPILOG Objects

These routines create formulas, predicates, operators, etc. Some are also described in the user manual.

(make-prop *neg &key quantifier variable restriction main-clause pred args*) [*function*]

This function creates a proposition (in list form). If the quantifier is given, the new proposition will be of the form *(quantifier variable restriction main-clause)* otherwsie it will be *((car args) pred (cdr args))* . If *neg* is non-nil, **not** is placed at the front of the proposition.

(make-fn-arg *fnname args*) [*function*]

creates a normalized functional term with function *fnname* and arguments *args* .

(new-record *contents &optional sort*) [*function*]

creates a normalized record term with sort *sort* and contents *contents* .

(normalize *formula &optional context insertion new-vars*) [*function*]

When calling from a specialist, only the *formula* parameter should be specified, and it should be a proposition in the form of a list. The result will be a normalized formula - an atom.

(normalize-wff *term &optional context*) [*function*]

This function takes a list form wff and normalizes it into an atom. No variable renaming is done, and trigger literals are not maintained.

(normalize-term *term &optional context*) [*function*]

This function takes a list form term and normalizes it into an atom.

(normalize-pred *pred &optional context*) [*function*]

This function takes a list form pred and normalizes it into an atom.

(normalize-op *op &optional context result*) [*function*]

This function takes a list form operator and normalizes it into an atom. *Result* is the expected type of the operator.

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

adds a new predicate. 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, but this serves to add type predicates quickly and easily for testing. Part predicates are specified when *part* is t. *Indicators* may be added here, or through the command **add-indicate** . 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 and entry routine to save information using this predicate, *entry-rtn* should also be specified. Some additional keys are *sort* and *specialists* .

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

adds a new function *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* .

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

adds a new operator. *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!

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

defines a new operator type. Operators of this type create objects of type *result-type* and expect 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.

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

where *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.

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

adds a new quantifier *quant* . 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 is the quantifier is distributive.

3.3 Matching

These routines are to be used to unify sets of arguments, and to make substitutions in propositions and terms. All are non-destructive - the unifications produce substitution lists and do not change the arguments. The substituion routines produce a new copy of the original with the substitutions made.

3.3.1 Unification

(compare-arglists *args-1 args-2 &key match-const-fn same-clause*) [*function*]

compare-arglists tries to find a unifying substitution between two argument lists, using the standard unification method. Arguments can be any lisp object, but the only variables and functions recognized are those that are concept nodes.

For the purpose of unification, propositional arguments are considered to be constants. It is possible to override what would normally be a constant mismatch by specifying **:match-const-fn** . The value of this keyword should be a function that accepts two arguments, and returns non-nil if the arguments should be considered equivalent for the purpose of unification.

If the same variable appears in both argument lists it is considered to be two distinct variables, unless the value of **:same-clause** is non-nil.

Normally, any object can be substituted for a variable, but variables which have been tagged with a sort can only be substituted by concepts with the same sort.

If a unifying substitution can be found, **compare-arglists** returns a dotted pair of substitution lists—one for each of the orginal argument lists. If unification fails, **compare-arglists** will return nil. It is possible for unification to succeed without substitutions, in which case **compare-arglists** will return **(nil . nil)** .

Each substitution in a substitution list is itself a three element list. The first element is the argument to be substituted, the second is its substitution, and the third is either **1** or **2** , indicating which argument list the substitution came from. (This last number is needed to distinguish between variables which appeared in both argument lists.) If the value of **same-clause** is non-nil, then all substitutions are considered to come from the first argument list.

To aid literal comparison, a unification routine was set up in the specialist, which takes two argument lists, an ordering list, and a last-chance test for unification. This routine may be used by the specialist to simplify unification.

(unify-lists *arglist1 arglist2 ordlist testrtn mult-subs*) [*function*]

where *arglist1* is the argument list from the first literal, and *arglist2* the argument list from the second literal. *Ordlist* is a list of dotted pairs of the form *(element number from 1st list . element number from 2nd list)* . For example ((1 . 2) (2 . 1)) would unify the 1st element of *arglist1* with the 2nd element of *arglist2* and the 2nd element of *arglist1* with the 1st element of *arglist2* ; in effect, reversing the order of the arguments. This routine actually calculates the new argument lists and then calls **compare-arglists** . *Mult-subs* indicates whether the substitutions should be applied recursively (i.e. if there is a substitution of x for y , and another of *c1* for x , with this flag, all y 's would end up as *c1* , without it, they end up as x .

Testrtn is a routine for the unifier to call when it has two constants to unify. Normally two constants don't unify, but in the case of specialists, we can still do something useful. *Testrtn* should have the calling sequence: *(testrtn arg1 arg2)* , and should return t if there is some known relation between *arg1* and *arg2* , and nil otherwise. If it is time consuming to determine the exact relation, the routine should return t if it is likely that there is a relation, rather than calculating it. If there really wasn't any point, that will be discovered in the actual resolution/factoring attempt anyway. Constants are not substituted for.

Unify-lists returns nil if no unification is possible, otherwise a dotted pair of the form returned by **compare-arglists** (consisting of substitutions for *arglist1* dotted with substitutions for *arglist2*).

(make-sub-element *newarg oldarg &optional which*) [*function*]

makes a substitution element consisting of the new argument and the old. If there is no change, nil is returned.

(arg-sub-list *newargs oldargs &optional which*) [*function*]

returns the substitution list required to make the list *oldargs* look like *newargs* . This is especially useful in a comparison attempt after unification and term evaluation have been done to add the necessary substitutions for the evaluated terms to the substitution lists returned with resolving actions. *Which* indicates which literal in the comparison attempt these arguments belong to (1 or 2).

unify-sorts [tweakable]

if this parameter is set to t (the default), the sorts of variables and constants will be taken into consideration during unification. Note however that a null sort will match any sort.

(print-subs *sublist*) [*function*]

returns a list of the form *((var by item) (var by item) ...)* which may be included in a **print-line** to print the substitutions in a nice format. The existing specialists use this method to print substitutions. For example, *(print-subs '((x c1 1) (y c2 1))* would return list *((x by c1) (y by c2))* .

(print-trace-subs *sublist*) [*function*]

actually prints the substitutions returned by unification. Printing is done on the current line so that it may be combined with other information. If there are no substitutions, "nothing" is printed, otherwise the substitutions are printed in the form *(sub/var sub/var ...)* . For example, *(print-trace-subs '((x c1 1) (y c2 1)))* would print:

c1/x c2/y

Typically other print statements will surround this to give more complete information:

with substitutions: c1/x c2/y in (x love y)

3.3.2 Formula Matching

(compare-wffs *wff1 wff2 compatible prefer*) [*function*]

compares two formulas, *wff1* and *wff2* to see if *wff2* is compatible with *wff1* (if *compatible* is t), or incompatible with *wf1* (if *incompatible* is nil). *Prefer* is optional, and if specified, must be either 1 or 2. If 1, it means that when unifying two variables from *wff1* and *wff2* , it is prefered that the variable from *wff1* be kept, and the substituted for the other one. **Compare-wffs** invokes specialists if applicable and if the appropriate flags are turned on. If successful, a list of *comparison-item* s will be returned, otherwise nil.

(incompatible-with-eval *test-item eval1 eval2 subs1 subs2 traced header incompatible*)[*function*]

After substitution and term evaluation have been done for two literals being resolved, this routine will create the resulting comparison items, and return them. It returns nil if the evaluations are not enough to indicate (in)compatibility. *Traced* indicates whether resolution tracing is on in the specialist, and *header* is a header for any messages this routine prints about adding subgoal agenda items. *Header* should be a string of the form " xxx Specialist:".

comparison-info [structure]

a list of these structures is returned by the comparison routines (such as **compare-wffs**), and expected to be returned by specialist comparison routines. To create one of these structures, use the following:

(make-comparison-info :wff1 *wff1* **:wff2** *wff2* **:lit1** *lit1* **:lit2** lit2

:subs1 *subs1* **:subs2** *subs2* **:residue1** *residue1* **:residue2** *residue* **)**

Residue1 and *residue2* are optional. Specialists needn't worry about *wff1* and *wff2* as they will only be working with the sub literals - a higher level routine will update those fields.
The fields are as follows:

comparison-info-wff1	the first formula in the comparison
comparison-info-wff2	the second formula in the comparison
comparison-info-lit1	the literal that was matched from *wff1*
comparison-info-lit2	the literal that was matched from *wff2*
comparison-info-subs1	substitutions required in literal 1
comparison-info-subs2	substitutions required in literal 2
comparison-info-residue1	residue for literal 1
comparison-info-residue2	residue for literal 2

3.3.3 Substitution

(substitute-prop *prop subs &optional subs-2 subs-num mult-subs*) [*function*]

(substitute-term *term subs &optional subs-2 subs-num mult-subs*) [*function*]

These two functions will perform a set of substitutions on propositions or terms. The format of the

substitution list *subs* is as described for **compare-arglists** . For each argument in the proposition or term, the substitution list is searched for a substitution triple whose first element is **eq** to the argument. If one is found, the argument is replaced by the second element of the triple. If *mult-subs* is true (if anything has been added using **arg-sub-lists** this should be true, otherwise nil), the replacement itself is then checked to see if any substitutions can be done within it, using the substitution list specified by the third element of the substitution triple—if **1** , the substitution list *subs* is used; if **2** , the substitution list *subs-2* is used. It's possible to omit the third element in a substitution triple, in which case the substitution list currently in effect will remain so. Normally, the *subs* substitution list is the first to be used, but this can be changed to be *subs-2* , by specifying *subs-num* to be **2** . Circular substitutions are not detected—the functions will simply recurse forever if given one.

The proposition or term passed to these functions and all of the substitution replacements which are nodes must be interned. The functions return interned nodes. Variables which are not substituted out will be left as is—no variable renumbering is done.

3.3.4 Hierarchies

(add-hier *args*) [*function*]

adds subnodes to a type or predicate hierarachy.

(add-part-hier *args*) [*function*]

adds subnodes to a part hierarachy.

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

sets the hierarchy type for *hier* . *Type* must be one of *(exclusion, overlap)* , or *pred-hier* .

(add-sort-to-hier *pred sort*) [*function*]

adds a sort to the type predicate *pred* so that any entity predicated with type *pred* or any predicate beneath it on a type hierarchy will be given the sort *sort=* .

(compare-hier-res *res1 res2*) [*function*]

returns the relationship between two entities in different hierarchies, given that *res1* is the relationship between the first entity and an entity common to both hierarchies, and *res2* is the relationship between that common entity and the other entity in question.

(hier-rel *a1 a2 &optional allowed-parts*) [*function*]

returns the relationship between entities *a1* and *a2* using the hierarchies. If *allowed-parts* is specified, the part hierarchies specified in that list may also participate in the determination of the relationship.

Chapter 4

EPILOG Specialists Interface Routines

This chapter contains the calling sequences and actions for a number of routines which enable a specialist to communicate with **EPILOG** and use its resources, and also to communicate with other specialists. A special interface called *spec* has been set up containing these routines, as well as the low level **EPILOG** routines described in the previous chapter. This minimizes the number of conflicts due to imported symbols. The routines are located in directory *epi/spec/* , and the command

(require 'spec "epilog:spec;spec")

will load them into your workspace.

It would be wise to do a *use-package* very early in the specialist, or simply to add *spec* to the package use-list when defining the package. For example:

(in-package 'time-specialist :nicknames '(time) :use-list '(lisp user spec lib))

4.1 Definition Routines

These commands are used to describe a specialist to **EPILOG** so that it knows when it is appropriate to call the specialist.

(set-specialist *specialist-name defn-file nicknames description*) [*function*]

tells **EPILOG** that a specialist by the name of *specialist-name* is available. When activated, the file *defn-file* will be loaded, which should contain a **define-specialist** command. *Nicknames* is optional, and is a list of other names that may be used to invoke the specialist. *Description* is a string indicating what the specialist's domain is. This command belongs in the *specialists* file in the *epi/* directory.

(define-specialist *specialist-name loadfile predicates operators sorts functions topics*)[*function*]

this command belongs in the definition file for a specialist which is loaded when a specialist is activated. *Specialist-name* is the name of the specialist, *loadfile* is the file which contains the code

for the specialist (and should start with a new package definition!), *predicates* are predicates which this specialist is interested in, *operators* are operators the specialist is interested in, and *functions* are functions the specialist is interested in. *Sorts* are the sorts that the specialist expects the first argument of a literal to have. This is an additional check after checking the predicate and operator to see if the specialist should be invoked. *Topics* is a list of default topics that predicates of the specialist should indicate.

The following routines should be used for specialists whose predicates are not all known at start-up time. The *add-predicate, add-operator* , or *add-function* routines described in the previous chapter may be used instead of these - the action will be identical.

(create-predicate *predicate specialist*) [*function*]

dynamically adds *specialist* to the property list of specialists for *predicate* , creating a concept for *predicate* if none existed. This requires that *specialist* has already been defined and activated, and is designed for those specialists whose predicates are not all known at start-up time. Note, you might also want to use **add-indicate** when you use this.

(create-operator *operator specialist*) [*function*]

dynamically adds *specialist* to the property list of specialists for *operator* , creating a concept for *operator* if none existed. This requires that *specialist* has already been defined and activated, and is designed for those specialists whose operators are not all known at start-up time.

(create-function *function specialist*) [*function*]

creates a concept of the appropriate form for functions if one didn't exist, and dynamically adds *specialist* to the property list of specialists for that concept. This requires that *specialist* has already been defined and activated, and is designed for those specialists whose functions are not all known at start-up time. It also declares *function* to be a function in the net.

(function-with-rel-pred *function pred &optional sort*) [*function*]

Sometimes it is desirable to have formulas containing functional terms "flattened" to conform more with specialist expectations. This function puts relational predicate *pred* and *sort* on the property list of *function* so that it be used in converting assertions of the form *((function a1) p1 a2)* into *(a1 pred c_sort)* and *(c_sort p1 a2)* . For example, if *((cardinality-of s1) less-than 3)* is asserted, it will not be sent to the set specialist, since the predicate is not a set predicate. When flattened, two formulas result, one for the set specialist,

(newsymbol_number less-than 3)

and one for the number specialist,

(s1 has-cardinality new-symbol) .

The set specialist can then ask the number specialist for more complete information.

4.2 Communication with Other Specialists

Specialist interface routines for handling immediate and delayed communication are as follows.

4.2.1 Immediate Evaluation

Sometimes information necessary for a specialist may not be available, but some other specialist might be able to help. The interface has been set up so that a specialist may ask for a particular functional term to be evaluated, or an entire literal. The request will be sent to any interested specialists, who may or may not provide an answer. The requesting specialist does not need to know which specialist(s) will answer his plea, or even if such a specialist exists.

(spec-eval-fn *fnarg*) [*function*]

immediately evaluates a functional argument. *fnarg* is the term to be evaluated. The specialist interface will locate applicable specialists and invoke them to evaluate the function. The result is returned by the function. If the functional term cannot be evaluated, *fnarg* itself will be returned. If this is an embedded function (some of the arguments are also functional terms), the result may be a simplified function, which still has not been evaluated.

(eval-with-spec *netname prop effort*) [*function*]

immediately evaluates a literal. *Netname* should be the current subnet identifier (this is a parameter available from the invokation of the particular specialist, but is also available in ***current-netname***), *prop* is the literal to be evaluated, and *effort* the amount of work to be used to evaluate it. The specialist interface finds the applicable specialists and invokes them, and returns the result of the evaluation. This is the same process used by **EPILOG** to find specialists to evaluate literals, but no "stripping" of subnet information is done here - it is assumed that specialists will not want to jump from one belief subnet to another.

(spec-eval-args *literal*) [*function*]

returns the list of arguments for *literal* after first trying to evaluate them. This is useful during the literal comparison phase of a specialist after unification has been done - any functions which can now be evaluted are evaluated here.

4.2.2 Delayed Communication

Occasionally information will not be available for an immediate evaluation, or the specialist may wish to be informed of any changes to that information so that it can keep its own representation up to date. To handle this, interested party lists keep track of entities which specialists are interested in, and literals which can be reasserted to them which will ensure that the desired information is retrieved and used properly. If a specialist makes a change to some internal information for a particular entity, it must notify the interface. When the specialist entries have finished, the list of changed entities is examined,

and if any have interested party lists, the appropriate specialists will be called. They can then ask for the updated information.

(add-interested-party *item spec*) [*function*]

adds an entry to the interested party list of *item* . This entry will consist of the specialist requesting it *spec* , the current literal being asserted, and the current subnet identifier. When any other assertions are made containing *item* with the same subnet identifier, *spec* will be called to resassert this literal.

(spec-changed-concept *concept*) [*function*]

indicates that some information about *concept* has changed, and adds it onto the list of concepts whose interested party lists are to be reasserted.

4.3 Communication with EPILOG

The communication from **EPILOG** to the specialists is all quite automated. Besides the evaluations that specialists communicate back to **EPILOG** , they may also make input-driven inferences, and send these back to **EPILOG** . A variable called ***assertions*** is available for any specialist to add inferences to. These inferences must be in unnormalized, list form. When the current input formula has been examined by all the interested specialists, and regular input-driven inferencing is about to take place, **EPILOG** looks at the list ***assertions*** and asserts each of them as if they were regular inferences.

assertions [variable]

a place to put input-driven inferences made by specialists so that they can be communicated back to **EPILOG** . ***assertions*** should be in the form of a list of non-normalized (i.e. list form) formulas.

spec-assert [tweakable]

is a tweakable parameter which indicates whether or not **EPILOG** should accept the specialist inferences on the ***assertions*** list. The default is t (accept).

4.4 EPILOG Usage of Specialists

This section describes the routines **EPILOG** uses to call specialists, and the expected top level definitions required by the specialists. Note that a user should NOT actually call these routines, except for testing purposes. The routines are called automatically by **EPILOG** . They are included here so that you can get a better understanding of how the interface works.

4.4.1 Specialist Activation

(use-spec *&rest specialists*) [*function*]

activates all the specialists in *specialists* . The specialists may be indicated by their full names or specified nicknames. If no specialists are included the ones remaining available that are not yet activated will be displayed.

available-specialists [variable]

includes the full names of all the specialists which can be activated.

active-specialists [variable]

includes the full names of the specialists which are currently active.

4.4.2 Specialist Invokation

(spec-compare-preds *pred1 pred2*) [*function*]

compares the two normalized predicates *pred1* and *pred2* using specialists if any are applicable. If an applicable specialist is found, and the routine *compare-preds* exists in the specialist's package, that routine is invoked as follows:

(compare-preds *pred1 pred2* **)**

where *pred1* and *pred2* are the predicates to be compared. These include any operators acting on them, and the specialists themselves are responsible for discarding any with operators which they do not handle. One of *disjoint* , *equivalent* , *subsumes* , *subsumed* , or *unknown* must be returned.

If more than one specialist is applicable, all are called until an answer other than *unknown* is returned.

spec-compare-preds [tweakable]

this flag indicates whether or not specialists should actually be used to compare predicates. If this flag is nil, **spec-compare-preds** will not be invoked, and no specialists will be used to compare predicates. Note that this should never be tweaked to nil, except for testing purposes.

(spec-enter *literal*) [*function*]

"flattens" *literal* if necessary, and strips off any modal embedding to determine subnets. It then determines the applicable specialist(s), and if a routine called *enter* is found in the specialist's package, it is called with the following:

(enter *netname lit neg pred arglist* **)**

where *netname* is the subnet indicator determined after any modal embedding is stripped off, *lit* is

the formula to be entered, *neg* indicates whether or not the formula is negated, *pred* is the predicate in the formula, and *arglist* are the arguments to the predicate. The specialist should return t if it entered something, and nil otherwise, but this is not required.

Note that ALL interested specialists are called to enter the information.

spec-enter [tweakable]

a tweakable flag that indicates whether or not specialists should be called to enter information in their own domains. The default is t (enter). If this is set to nil, **spec-enter** will not be called, and no formulas will be entered into specialists' representations - thus no evaluations with specialists that normally save formulas themselves will be possible.

specialist-entry-effort [tweakable]

the effort level that a specialist should use when entering a formula to determine how hard it should work at guaranteeing consistency. 0 indicates constant time only operations. Any number higher than that is specialist specific as to the effect it has.

(spec-evaluate *literal &optional effort*) [*function*]

like **spec-enter** , the formula is first flattened if necessary, and has any modal embedding stripped off. Interested specialists who have a routine called *evaluate* in their packages are called in the following way:

(evaluate *netname lit neg pred arglist effort* **)**

where *netname* is the subnet indicator determined after stripping off modal embedding, *lit* is the formula to be evaluated, *neg* indicates whether or not the formula is negated, *pred* is the formula's predicate, and *arglist* are the arguments to the predicate. *Effort* indicates how hard the specialist should work to evaluate the formula. This usually defaults to **specialist-eval-effort** . The evaluation routine should return one of *yes* , *no* , or *unknown* .

Applicable specialists are called until one returns an answer other than *unknown* .

spec-evaluate [tweakable]

indicates whether or not specialists should be used to help evaluate formulas. The default is t (let them help). If tweaked to nil, **spec-evaluate** will not be called, and so no specialists will be invoked to evaluate a formula.

specialist-eval-effort [tweakable]

the effort level that a specialist should use when evaluating a formula. 0 indicates constant time only operations. Any number higher than that is specialist specific as to the effect it has.

(spec-eval-fn *fnarg*) [*function*]

determines specialists interested in *fnarg* based on the function name alone, and if any specialist contains a routine the same as the function name in its package, that routine is invoked with:

(apply *function args* **)**

where *args* are the arguments of the functional term *fnarg* . The function should either return a normalized term, or nil. If more than one specialist contains such a function, they are all called until a non-nil answer is returned.

spec-eval-fn [tweakable]

indicates whether or not functions should be evaluated by specialists, using the **spec-eval-fn** routine. Currently the flag is not checked - specialists are allowed to evaluate the functions. This should probably be fixed to make it more flexible (and for testing purposes).

(spec-incompatible-lits *lit1 lit2 &optional effort*) [*function*]

(spec-compatible-lits *lit1 lit2 &optional effort*) [*function*]

first strip off any modal embedding and ensures that both formulas refer to the same subnet. Then they determines specialists which are interested in BOTH *lit1* and *lit2* , and if any contain a routine called *incompatible-lits* (for **spec-incompatible-lits**) or *compatible-lits* (for **spec-compatible-lits**) in their package, the routine is called in the following manner:

(incompatible-lits *netname lit1 lit2 neg1 pred1 arglist1 neg2 pred2 arglist2 effort* **)**

(compatible-lits *netname lit1 lit2 neg1 pred1 arglist1 neg2 pred2 arglist2 effort* **)**

where *netname* again indicates the subnet after modal embedding has been stripped off, *lit1* and *lit2* are the formulas to be compared, *neg1* and *neg2* indicate whether the *lit1* and *lit2* are negated, respectively, *pred1* and *pred2* are the literal's predicates, and *arglist1* and *arglist2* are the argument lists. *Effort* is an indicator of how hard the specialist should work at comparing the literals. This is usually defaulted to *specialist-eval-effort*. The routine should return a list of *comparison-item* s if it is successful, nil otherwise.

Note that ALL applicable specialists are called.

spec-compare-lits [tweakable]

indicates whether or not specialists should be allowed to participate in literal comparison. This is initially set to nil (do not let them participate) because the operation is usually expensive and is not always helpful. More investigation is required into this. If nil, **spec-incompatible-lits** and **spec-compatible-lits** will never be called.

fwd-spec-compare [tweakable]

An additional flag to control the specialist comparison of literals. If specialist comparisons are allowed, they are usually useful for goal-directed reasoning (questions), but rarely for input-driven inferencing. If this flag is tweaked off, specialists will NOT be used to compare literals during forward inference, even if ***spec-compare-lits*** is t. If tweaked on, ***spec-compare-lits*** makes the decision.

Index