FRIL Help Documentation Rewritten

March 2, 2020

1 help

Valid options are as follows:
help <**pred_name**>
help bips
help trace
help notation
help prologs
help modules
help menus
help dialogs
help limits
help errors
help help
help online
help preferences

2 bips

Information on built-in predicates is provided in sections as follows:
help control
help support
help arith
help input
help output
help files
help process
help system
help fuzzy
help kb
help unification
help types
help conversion

help strings
help windows
help menu_preds
help module_preds
help debugging
help dialog_preds

3 trace

The trace package is invoked using the system predicate "tq", in place of "?". "tq" follows exactly the same "depth first" execution strategy that Prolog uses, but it allows the user to observe the results of instantiations of successful goals and provides a certain degree of interaction and control over the quantity of trace information which is displayed and how the execution of goals proceeds.
The FRIL tracer is stored as a compiled module in the Fril library, and it is loaded automatically whenever it is required. This is totally transparent to the user who can regard the trace as part of the Fril system.
Spy points can be set and cleared on particular predicates, and the leap mode can be used to suppress trace messages from all predicates other than those being spied. At any time the trace can be temporarily suspended to a Trace-break level which simulates a Fril top-level interpreter. From here arbitrary Fril commands can be executed at this level, including adding procedures to the database. For further details type "help tq" or use the on-line help for built-in predicates in the category "Debugging".

4 modules

A Fril module is a compiled set of clauses that can be loaded or killed as a single unit. Because the module is stored in compiled form, it can be loaded much faster than the corresponding source code. Furthermore, since most of the module's internal code is inaccessible to the rest of the program, Fril can make more radical optimisations than are usually possible.
Modules are particulary useful in the design of large programs, as each section can be treated as a "black box", with a specified interface to the rest of the program.
Predicates within a module are assumed to be only called from within that module, and not to call predicates outside that module (there are aditional assumptions; see section xxx of the manual). The interface is specified by declaring as "import", "export", "dynamic" or "visible" those predicates which do not obey the above assumptions.
In order to be built into a module, a file has to be passed through the module compiler using the built-in predicate "module". For further information, request

help on
"module", "module_initialisation", "import", "export", "dynamic" and "visible". Type
help module_preds
for a list of built-in predicates related to modules.

5 dialogs

A Fril dialog is a user-defined dialog window containing various controls. There are two kinds of dialog, modeless dialogs and modal dialogs.
Modal dialogs suspend all execution and dominate the system until dismissed. No other processing is performed, and it is not possible to switch to other applications. Once the dialog has been dismissed, Fril carries on executing.
A modeless dialog displays a window which can be moved, hidden, etc. like an ordinary window; once it has been displayed, Fril continues execution. If a button, check box, radio button, or popup menu is clicked, a Fril goal is executed.
The Fril Dialog compiler is a utility for creating arbitrary dialogs in Fril. Type
help dialog_preds
for a list of built-in predicates related to dialogs.

6 menus

Fril allows items to be added to the menu bar under program control. The command
menu <**const**>
prior to clauses defining the predicate <**const**> causes <**const**> to be added to the menu bar. Clauses for <**const**> should have an arity of one and each clause in the definition should have a different constant as its argument. Each constant appears as an option under the menu. Selecting an option is the same as executing the goal (<**const**> <**option**>)from Fril top level.
If a fact in the definition of a menu predicate has a single constant argument beginning with a hyphen, e.g.
((Test -))
then a dotted line is drawn across the menu. Type
help menus
for a list of built-in predicates related to menus.

7 foreign

The Fril foreign language interface is now available as a separate product, and is not documented in this help system.

8 notation

Many built-in predicate arguments are restricted to a particular form. In Chapter 7 of the reference manual and in the on-line help summaries of built-in predicates, arguments are written in angle brackets to indicate such a restriction, e.g. <**int**> denotes that an argument must be an integer. A numerical suffix is used to distinguish restricted arguments of the same type, e.g. <**int1**>, <**int2**>. Many of the restricted forms are defined in Chapter 7, but all forms are displayed below:
<**asc_int**> an integer between 0 and 252, corresponding to a character code
<**atom**> a list.
<**atom_list**> a list of atoms.
<**body**> a list of atoms.
<**char**> a Fril constant whose print-name is a single character.
<**cl_supp**> either a <**support**> or a list of two <**supports**>.
<**clause**> a Fril clause, i.e. a list whose first element is a list with a constant predicate name as first element. The most general form for <**clause**> is ((<**const**> |<**var1**>) |<**var2**>)
<**const**> a Fril constant.
<**dtype**> a Fril dtype (discrete fuzzy set).
<**filename**> a Fril constant which names a file.
<**float**> a floating point decimal number.
<**fset**> a Fril itype (fuzzy number), dtype (discrete fuzzy set), or universe.
<**head**> a list whose first element is a constant predicate name.
<**in_stream**> a Fril constant which names a currently open input stream.
<**io_stream**> a Fril constant which names a stream currently open for input OR output.
<**int**> an integer.
<itype a Fril itype (fuzzy number).
<**named_dtype**> a constant naming a Fril dtype.
<**named_itype**> a constant naming a Fril itype.
<**nni**> a non-negative integer.
<**num**> an integer or floating point decimal number.
<**out_stream**> a Fril constant which names a currently open output stream.
<**rgb**> a list of three integers in the range 0-65535, (red, green, blue components of a colour).
<**support**> either a number in the range 0 to 1, or a list of two numbers in the range 0 to 1.
<**term**> any valid Fril object, i.e. a list, number, constant, variable or fuzzy set.
<**univ**> a Fril universe.
<**var**> a Fril variable.
<**var_support**> either a variable, or a list of two variables, e.g. S or (L U).

In Chapter 9 and in the online help summaries, the notation "pred_name"/<arity> is used to denote that the predicate called "pred_name" has <**arity**> number of arguments. <**arity**> may be one or more non-negative integers, or the symbol 'n' to denote variable arity, or for example 2+n to denote at least 2 but otherwise variable arity, e.g.
repeat/0 sum/3
addcl/1,2,3,4 peek, peekb /2
or/n isall/2+n

9 prologs

Fril contains a complete Prolog system, but is considerably enhanced by its facilities for handling uncertainty. If these features are not used, Fril can be regarded as a dialect of Prolog. In this case, the most obvious difference between Fril and other Prolog systems is the syntax, since most Prolog systems use a functional notation for goals e.g. <**pred**> (<**arg1**>, **argn**>) whereas Fril would use (<**pred**> <**arg1**>,**argn**>). Apart from the other enhancements arising from the support logic system, the main differences between Fril and other Prologs, are in the built-in predicates and handling of meta-variables.
"delcl" corresponds to "retract" of Edinburgh syntax Prologs, in that it deletes a clause from the knowledge base. In contrast to "retract", however, delcl is NOT resatisfiable.
Edinburgh syntax "bagof" is implemented as "findall" (or isall). Variables which appear in the goal to be solved, but which are not in the answer pattern, are treated as existentially quantified. In contrast to "bagof", "findall" (and isall) are not resatisfiable.
"not" in Fril is a support logic negation, rather than an implementation of negation-as-failure. The built-in predicates "neg" and "negg" perform negation-as-failure.
Access predicates such as "functor", "=..", "arg", etc. are unnecessary in Fril due to its list based syntax - the relevant terms can be accessed by standard list-processing methods.
See also section 1.xx of the reference manual.

10 errors

Not yet included (sorry!)

11 preferences

Not yet included (sorry!)

12 limits

The maximum length of a constant or variable name is 126 characters.
The maximum arity of a relation is 31, and all tuples in a relation must be of the same arity. The maximum size of a compiled tuple is 4072 bytes.
Each element in an itype must be a number. Each element in a dtype fuzzy set must be aconstant or number. The membership level of each element in the itype must be a number in the range 0 to 1.
The following values are dependent on the host machine and operating system:
The maximum stack and knowledge base size,
The maximum values of real and integer numbers,
The maximum number of files open for input or output.
Typical values are as follows:
Stack, knowledge base ..... dependent on available core.
Real range ..... approx 3.0E38 to 3.0E-38
Integer range ..... -2147483648 to 2147483647
The maximum number of user files is 17.

13 control

!/0 Cut, prevents backtracking
?/1 Query, executes program
abort/0 aborts execution and resets system to top level
errm/1 switches or returns control error message status
error/6 default system error handler
err_han/1 error handler - redirection of control on recognition of errors
exit/1 terminates execution
fail/0 forces backtracking
findall/3 returns all solutions to a query, in a list
forall/2 checks that a condition holds for all solutions to a goal
general/2 support logic meta-predicate for implementing extended fril rules
if/3 simulates if hen lse
isall/2+n returns all solutions to a query, in a list (similar to "findall", but with different syntax)
module_initialisation/1 specify goals to be executed when a module is loaded
neg/1 negation as failure
negg/n negation as failure
oh/1 finds and prints one solution to a query, backtracking to find another solution on request

orr/n standard prolog disjunction
os/1 finds and prints support for one solution to a query, backtracking to find another solution on request
qh/1 finds and prints all solutions to a query
qs/1 finds support for all solutions to a query, and prints each solution and support
repeat/0 succeeds repeatedly (always resatisfiable)
sc/1 execute goal list generating weighted mean of expected values
se/1,2 execute goal list generating expected fuzzy sets
snips/1+n makes a goal succeed only once
supp_collect/2 executes query and calculates weighted mean of expected values from combined fuzzy sets for each conclusion
supp_expect/2,3 executes program and calculates support for each conclusion, combining fuzzy sets into an expected fuzzy set where appropriate
supp_query/2,3 executes program and calculates support for each conclusion
supp_value/2,3 executes program and calculates support and expected value from combined fuzzy sets for each conclusion.
sv/2, 3 execute goal list generating expected values
wh/1 finds all solutions to a query and prints an answer pattern for each
ws/1 finds support for all solutions to a query and prints an answer pattern and support for each

14 support

and/n default support logic conjunction
ask_support/3 request a support pair using a built-in dialog
conj/n support logic conjunction, with no dependency assumptions
dempdict/1 generate or check constants which name a dempster procedure in the knowledge base
dempster/1 define a dempster procedure
disj/n support logic disjunction, with no dependency assumptions
evlog/1 support logic meta-predicate for implementing evidential logic rules
expected_value/2 computes expected value of a fuzzy set
gen/3 support logic meta-predicate for implementing extended fril rules with non-standard conjunction
general/2 support logic meta-predicate for implementing extended fril rules
match/2 compute the support for the semantic unification of two terms
not/1+n support logic negation
or/n default support logic disjunction
os/1 finds and prints support for one solution to a query, backtracking to find another solution on request
osc/1 finds and prints support and (if appropriate) weighted mean of expected values for one solution to a query, backtracking to find another solution on request

ose/1 finds and prints support and (if appropriate) expected fuzzy set for one solution to a query, backtracking to find another solution on request
osv/1 finds and prints support and (if appropriate) expected value for one solution to a query, backtracking to find another solution on request
point_match/2 compute the point value support for the semantic unification of two terms
poss_match/2 compute the support for the possibilistic semantic unification of two terms
qs/1 finds support for all solutions to a query, and prints each solution and support
qsc/1 finds and prints support and (if appropriate) weighted mean of expected value for all solutions to a query
qse/1 finds and prints support and (if appropriate) expected fuzzy set for all solutions to a query
qsv/1 finds and prints support and (if appropriate) mean of expected fuzzy set for all solutions to a query
sc/1 execute goal list generating weighted mean of expected values
se/1,2 execute goal list generating expected fuzzy sets
supp_collect/2 executes query and calculates weighted mean of expected values from combined fuzzy sets for each conclusion
supp_expect/2,3 executes program and calculates support for each conclusion, combining fuzzy sets into an expected fuzzy set where appropriate
supp_query/2,3 executes program and calculates support for each conclusion
supp_value/2,3 executes program and calculates support and expected value from combined fuzzy sets for each conclusion.
sv/2,3 execute goal list generating expected values
system_garbage/0 remove system modules and intermediate clauses generated by expected fuzzy set and/or expected value predicates
true/1 simulates solution of a goal with a specified support
ws/1 finds support for all solutions to a query and prints an answer pattern and support for each
wsc/1 finds support and (if appropriate) weighted mean of expected values for all solutions to a query and prints an answer pattern and support for each
wse/1 finds support and (if appropriate) expected fuzzy set for all solutions to a query and prints an answer pattern and support for each
wsv/1 finds support and (if appropriate) expected value for all solutions to a query and prints an answer pattern and support for each

15 arith

convert/2 convert between numbers and their string representations
cos/2 cosine and arccosine
division/3 integer division
eq/2 unify two terms

exp/2 natural exponential and logarithm
init_random/1 initialise the random number generator
int/1 test whether a term is an integer
less/2 comparison of numbers or constants
less_eq/2 comparison of numbers or constants
num/1 test whether a term is a number
power/3 raising a number to a power
random/1 random number between 0 and 1
remainder/3 remainder on integer division
sin/2 sine and arcsine
square/2 square and positive square-root
stricteq/2 check equality of two terms (without unification)
sum/3 addition and subtraction
tan/2 tangent and arctangent
times/3 multiplication and division
truncate/2 truncates a floating point number to the nearest integer below

16 input

ask/3 request a string using a built-in dialog
ask_support/3 request a support pair using a built-in dialog
await_input/1 returns the next input event which can be handled by a Fril program
close/1 close a file
confirm/1 built-in dialog with yes and no buttons
create_r/1 open file for input and output, overwrites existing file
create_ra/1 open file for input and output, appends to existing contents of file
cscanf/2+n formatted input from a stream
edit/1 allows program control of some operations on the edit menu
filebcopy/3 file to file block copy
filepos/2 random access file position
filesearch/3 search through file for string
flush/1 flush input or output buffer
get/2 character input from stream
getb/2 character input from stream (including whitespace)
getk/1 immediate version of "getb" character input, from standard input
get_dlog_val/4 report status of a control item on a dialog
highlight/2 set or find the current highlighted selection in the topmost window
intok/2 tokenised input from stream
open/1 open a file for input
peek/2 examination of next character in stream input buffer
peekb/2 examination of next character in stream input buffer (including white space)

peekk/1 immediate version of "peekb" character examination, from standard input
r/1 read a term from standard input
read/2 read a term from a stream
read_suppterm/3 read a term and associated support from a stream

17 output

available_font/1 checks or generates names of fonts on the current system
close/1 close a file
cprintf/2+n formatted output to a stream
create/1 open file for output, overwrites existing file
create_a/1 open file for output, appends to existing contents of file
create_r/1 open file for input and output, overwrites existing file
create_ra/1 open file for input and output, appends to existing contents of file
default_format/2 set the default text format for new windows
edit/1 allows program control of some operations on the edit menu
filebcopy/3 file to file block copy
flush/1 flush input or output buffer
format/2 change or report current text format
highlight/2 set or find the current highlighted selection in the topmost window
p, pp, pq, ppq/n print terms on standard output
prlen/2 returns the print length of Fril terms
pspaces/1 print spaces on the standard output
putb/2 character output
scroll_top_window/1 scroll the text display in the top windos
w, wq/1+n print to an output stream, separated by spaces
write, writeq/1+n printto an output stream, separated by lines
wspaces/2 print spaces to an output stream

18 files

cd/1 change or report the current directory
close/1 close a file
cprintf/2+n formatted output to a stream
create/1 open file for output, overwrites existing file
create_a/1 open file for output, appends to existing contents of file
create_r/1 open file for input and output, overwrites existing file
create_ra/1 open file for input and output, appends to existing contents of file
cscanf/2+n formatted input from a stream

current_edit/1 report or change the topmost edit window
edit/1 allows program control of some operations on the edit menu
editdict/1 generate or check constants which name open edit windows
exists/1 checks whether file already exists
file/2 allows program control of some operations on the File menu
filebcopy/3 file to file block copy
filename/2 converts between full filename and directory, basename and extension
filepos/2 random access file position
filesearch/3 search through file for string
flush/1 flush input or output buffer
get/2 character input from stream
getb/2 character input from stream (including whitespace)
get_filename/1 get a filename using a standard file selection dialog
highlight/2 set or find the current highlighted selection in the topmost window
intok/2 tokenised input from stream
open/1 open a file for input
peek/2 examination of next character in stream input buffer
peekb/2 examination of next character in stream input buffer (including white space)
read/2 read a term from a stream
read_suppterm/3 read a term and associated support from a stream
tempfile/1 generates a temporary file name

19 process

cputime/2 time used by process
help/1 get help on built-in predicates and other topics
kb_garbage/0 removes garbage from knowledge base
statistics/0 resources used by process
system_garbage/0 remove system modules and intermediate clauses generated by expected fuzzy set and/or expected value predicates

20 system

cd/1 change or report the current directory
date/2 returns current date and time
exists/1 checks whether file already exists
exit/1 terminates execution
filename/2 converts between full filename and directory, basename and extension
getenv/2 gets value for environment variable

get_filename/1 get a filename using a standard file selection dialog
setenv/2 sets value for environment variables
sh/1 execute shell command
tempfile/1 generates a temporary file name

21 fuzzy

complement/2,3 find the complement of a fuzzy set with respect to its universe
ddict/1 generate or check named dtypes in the knowledge base
def_dtype/2,3 define a named dtype
def_itype/2,3 define a named type
dtype/1 check whether a term is a dtype
dtype_name/2 converts a dtype name to its fuzzy set representation and vice-versa
dtype_to_list/2 converts between a dtype and a list of element/membership pairs
fsetdict/1 generate/check any named fuzzy set in the knowledge base
fset/1 check whether a term is an itype or dtype
fset_name/2 converts a dtype, itype, or universe name to its fuzzy set representation and vice-versa
get_univ/2 find the universe associated with a fuzzy set
idict/1 generate or check named itypes in the knowledge base
intersection/3 intersection of two fuzzy sets
itype/1 test whether a term is an itype
itype_name/2 converts an itype name to its fuzzy set representation and vice-versa
itype_to_list/2 converts between an itype and a list of element/membership pairs
set/1 define a universe (domain) for fuzzy sets
set_difference/3 set difference of two fuzzy sets
sum/3 addition and subtraction
times/3 multiplication and division
union/3 union of two fuzzy sets
univ/1 check whether a term is a universe
univdict/1 generate or check universes defined in the knowledge base
univ_name/2 converts a universe name to its set representation and vice-versa

22 kb

addcl/1,2,3,4 add a clause to the knowledge base
cdict/1 generate or check constants which name a procedure in the knowledge base

cl/1,2,3,4 generate or check standard clauses, dempster clauses, menu clauses, dialog clauses, or tuples in the knowledge base
ddict/1 generate or check named dtypes in the knowledge base
def/1 checks whether a procedure or goal is defined
def_dtype/2,3 define a named dtype
def_itype/2,3 define a named type
def_rel/2 define a constant to be a relation name
delcl/1, 2 delete a clause from the knowledge base
dempdict/1 generate or check constants which name a dempster procedure in the knowledge base
dict/1 generate or check constants which name a procedure in the knowledge base
dlogdict/1 generate or check constants which name a dialog procedure in the knowledge base
dtype/1 check whether a term is a dtype
dynamic/1 declaration of procedures which may be modified by calls within the module
edict/1 generate or check export predicates
export/1 declaration of user accessible procedures in a module
fsetdict/1 generate / check any named fuzzy set in the knowledge base
fdict/1 generate / check foreign language procedures
idict/1 generate or check named itypes in the knowledge base
itype/1 test whether a term is an itype
kb_garbage/0 removes garbage from knowledge base
kill/1 delete a module, fset, or an entire procedure from the knowledge base
list/1 display procedures or fset definitions at the standard output
listfile/2 display procedures or fset definitions to a specified stream
lload/1 loads a list of files
load/1 take input from a file, or load a Fril module - usually used to add to the knowledge base
lreload/1 reloads a list of files
mdict/1 generate or check constants which name a currently loaded module
menudict/1 generate or check predicates on the menu bar
module/2 create a new module file from a specified source file of Fril program text
rdict/1 generate or check constants which name a relation in the knowledge base
reload/1 similar to load, but overwrites existing definitions in the knowledge base
set/1 define a universe (domain) for fuzzy sets
sys/1 check built-in predicate name
system_garbage/0 remove system modules and intermediate clauses generated by expected fuzzy set and/or defuzzification predicates
univ/1 check whether a term is a universe
univdict/1 generate or check universes defined in the knowledge base

23 unification

eq/2 unify two terms
match/2 compute the support for the semantic unification of two terms
point_match/2 compute the point value support for the semantic unification of two terms
poss_match/2 compute the support for the possibilistic semantic unification of two terms
stricteq/2 check equality of two terms (without unification)

24 types

atomic/1 test whether a term is a constant or number
con/1 test whether a term is a constant
dlogdict/1 generate or check constants which name a dialog procedure in the knowledge base
dtype/1 check whether a term is a dtype
dtype_name/2 converts a dtype name to its fuzzy set representation and vice-versa
eq/2 unify two terms
fsetdict/1 generate / check any named fuzzy set in the knowledge base
fset/1 check whether a term is an itype or dtype
int/1 test whether a term is an integer
itype/1 test whether a term is an itype
itype_name/2 converts an itype name to its fuzzy set representation and vice-versa
menudict/1 generate or check predicates on the menu bar
num/1 test whether a term is a number
stricteq/2 check equality of two terms (without unification)
sys/1 check built-in predicate name
univ/1 check whether a term is a universe
var/1 test whether a term is a variable

25 conversion

charof/2 convert between ASCII codes and characters
convert/2 convert between numbers and their string representations
dtype_name/2 convert a dtype name to its fuzzy set representation and vice-versa
dtype_to_list/2 convert between a dtype and a list of element/membership pairs
fset_name/2 convert a dtype, itype, or universe name to its fuzzy set representation and vice-versa

itype_name/2 convert an itype name to its fuzzy set representation and vice-versa
itype_to_list/2 converts between an itype and a list of element/membership pairs
name/2 convert between a list of ASCII codes and a constant
stringof/2 convert between a list of characters and a constant
str_to_list/2 convert between a constant and a list of terms
truncate/2 truncate a decimal number to the nearest integer below
univ_name/2 convert a universe name to its set representation and vice-versa

26 strings

atomic/1 test whether a term is a constant or number
charof/2 convert between ASCII codes (or extended codes) and characters
con/1 test whether a term is a constant
convert/2 convert between numbers and their string representations
dtype_name/2 converts a dtype name to its fuzzy set representation and vice-versa
fset_name/2 converts a dtype, itype, or universe name to its fuzzy set representation and vice-versa
gensym/2 generates a new constant symbol
itype_name/2 converts an itype name to its fuzzy set representation and vice-versa
less/2 comparison of numbers or constants
less_eq/2 comparison of numbers or constants
name/2 converts between a list of ASCII codes (or extended codes) and a constant
stringof/2 converts between a list of characters and a constant
str_to_list/2 converts between a constant and a list of terms
univ_name/2 converts a universe name to its set representation and vice-versa

27 windows

available_font/1 checks or generates names of fonts on the current system
back_col/1 change or report the background colour of the current window
clear_current_window/0 erase the contents of the current window
close_dialog/1 dismiss a user-defined modeless dialog
create_window/7 create a new Fril window
current_edit/1 report or change the topmost edit window
current_window/1 report or change the topmost Fril window
cursor_position/2 report or change the cursor position in the topmost Fril window

default_format/2 set the default text format for new windows
delete_window/1 remove a window
edit/1 program control of some edit menu operations
editdict/1 generate/check constants which name open edit windows
format/2 change or report current text format
getk/1 immediate version of "getb" character input, from standard input
hide_window/1 hide a window
highlight/2 set/find current highlighted selection in the topmost window
intok/2 tokenised input from stream
maximise_frame/1 expand root window to full screen size on a Windows 3.1 platform
max_window_pos/2 report the maximum x and y co-ordinates for a window position on the screen
max_window_size/2 report the maximum window size on the screen
min_window_pos/2 report the minimum x and y co-ordinates for a window position on the screen
min_window_size/2 report the minimum window size on the screen
move_dialog/1 report or change the screen location of a user-defined modeless dialog
move_window/3 change or report the position of a window on the screen
p/n print terms on standard output separated by spaces, no quotes
peekk/1 immediate version of "peekb" character examination, from standard input
pen_col/1 report or change the text colour in the current Fril window
pose_dialog/2 display a user-defined dialog
resize_window/3 change the size of a window
scroll_top_window/1 scroll the text display in the top windos
stack_windows/0 move and resize all windows
top_window/1 report or change the topmost window on the screen
wdict/1 generate or check constants which name a Fril window
zoom_top_window/1 expand window to maximum size, or restore to original size

28 menu_preds

disable_menu/1,2 dim an entire menu or a single item on the menu
enable_menu/1,2 enable an entire menu or a single item on the menu
menu/1 define a predicate to appear on the menu bar
menudict/1 generate or check predicates on the menu bar

29 module_preds

module/2 create a new module file from a specified Fril source file.
export/1 declaration of user accessible procedures.
import/1 declaration of procedures defined outside a module.
dynamic/1 declaration of procedures which may be modified by calls within the module.
visible/1 declaration of procedures which may be accessed by program meta-calls.
edict/1 generate or check export predicates

30 debugging

nospy/1 remove spy point(s) from predicate(s)
spy/1 set spy point on predicate
spying/1 generate or check which predicates are currently being spied
tq/1 Fril prolog trace query evaluator

31 dialog_preds

ask/3 request a string using a built-in dialog
ask_support/3 request a support pair using a built-in dialog
await_input/1 returns the next input event which can be handled by a Fril program
close_dialog/1 dismiss a user-defined modeless dialog
confirm/1 built-in dialog with yes and no buttons
dialog/1 specify a user-defined dialog
disable_dlog_item/3 dim a control on a dialog
dlogdict/1 generate or check constants which name a dialog procedure in the knowledge base
enable_dlog_item/3 enables a control on a dialog
get_dlog_val/4 report status of a control item on a dialog
get_filename/1 get a filename using a standard file selection dialog
inform/1 built-in dialog with an OK button
move_dialog/1 report or change the screen location of a user-defined modeless dialog
pose_dialog/2 display a user-defined dialog
set_dlog_val/4 change status of a control item on a dialog

32 !

Usage: (!)
"!", the cut, is used to alter the normal Fril execution strategy by removing alternative clauses. In an ordinary query, using the depth-first execution strategy, the cut removes all backtrack points between itself and the head of the clause in which it appears. When used in a support query, the cut removes alternatives, unless a goal preceding it in the clause has been proved false, i.e. has support (0 0).

33 ?

Usage: (? <**atom_list**>)
"?" executes the series of calls in <**atom_list**>, using the standard depth-first search mechanism. "?" can be nested within calls to "?" or other built-in predicates that take atoms or atom lists as arguments, e.g. neg, negg, orr, supp_query. Note that "?" appearing within a "supp_query" causes the search mechanism to change from breadth to depth-first until the "?" is satisfied.

34 abort

Usage: (abort)
"abort" terminates execution of the current goal, and resets the system to Fril top level. All output buffers are flushed, "errm" and "err_han" are reset, and any open streams are closed.

35 addcl

Usage: (addcl <**clause**>)
(addcl <**clause**> : <**cl_supp**>)
(addcl <**clause**> <**int**>)
(addcl <**clause**> : <**cl_supp**> <**int**>)
"addcl" adds <**clause**> to the knowledge base, with support <**cl_supp**> if given. By default, <**clause**> is added after any other definitions for the same predicate. Exceptions to the default are as follows:
(i) If the predicate names a clause set, then the third and fourth forms of addcl allow clauses to be added after the <**int**>'th existing clause. For instance, a value of 0 for <**int**> would make <**clause**> the first entry in the clause set for the predicate. If <**int**> is less than zero or greater than the number of clauses in the definition, then the new clause is added before the first or after the last existing clause, respectively.

(ii) If the predicate names a relation, then the clause (which must be a fact) is added as a tuple to the relation. Since tuples of a relation are held in an internally determined form, the <**int**> in the third or fourth form of addcl is ignored.

36 and

Usage: (and <**atom1**> ... <**atomn**>)
"and" implements the default Fril support logic conjunction. It accumulates the support for successive pairs of the support logic goals <**atom1**> to <**atomn**> combining them using the default support logic calculus for conjunction.

37 ask

Usage: (ask <**const1**> <**const2**> <**var**>)
"ask" displays a simple dialog box with a question, an editable text box for a response, and "OK" and "Cancel" buttons. <**const1**> is the text of the question, and <**const2**> is the default answer displayed in the response box. If the "OK" button is clicked, <**var**> is bound to the string in the response box.

38 ask_support

Usage: (ask_support <**const**> <**var1**> <**var2**>)
"ask_support" displays a dialog box with a question and two sliders for setting a support pair. Two editable text boxes give an alternative means of setting the supports. The dialog box has "OK" and "Cancel" buttons, and enforces the constraint that the lower support cannot exceed the upper support. If the "OK" button is clicked, <**var1**> and <**var2**> are bound respectively to the lower and upper supports returned by the dialog.

39 atomic

Usage: (atomic <**term**>)
"atomic" tests whether its argument is a constant or a number.

40 available_font

Usage: (available_font <**const**>)
(available_font <**var**>)

"available_font" checks whether a given font is available, or generates the name of a font which is available on the current system.

41 await_input

Usage: (await_input <**var**>)
"await_input" returns the next available input event. An input event is either
(i) a character from the keyboard when a Fril window is active,
(ii) a mouse click on a user-defined menu in the menu bar,
(iii) a mouse click in a user-defined modeless dialog box.
The goal succeeds as soon as the event happens, and keyboard input does not have to wait for a return key to be pressed.

42 back_col

Usage: (back_col <**term**>)
"back_col" reports or changes the background colour of the current Fril window. The new colour is represented by <**term**>, as described in Section 2.1. A background colour is associated only with the topmost Fril window at the time the goal is called; "back_col" will not affect the background colour of any other windows.

43 cd

Usage: (cd <**const**>)
(cd <**var**>)
cd reports the current directory in <**var**>, or changes it to the directory specified by <**const**>. <**const**> may be a relative or full directory pathname, and if any special characters such as 'áre included, then the whole directory specification must be formulated as a string by enclosing it in quotation marks.

44 cdict

Usage: (cdict <**term**>)
cdict checks whether <**term**> names a clause set in the knowledge base. If <**term**> is a variable, cdict binds it to the name of a clause set, assuming at least one clause has been defined. Different predicates can be generated by backtracking to cdict. Dempster, menu, and dialog definitions are not included.

45 charof

Usage: (charof <**char**> <**int**>)
(charof <**char**> <**var**>)
(charof <**var**> <**asc_int**>)
"charof" converts ASCII values to characters and vice-versa. The first form checks that <**char**> has the ASCII code <**int**>. The second form binds <**var**> to the ASCII code of the character <**char**>, and the third form binds <**var**> to the constant whose print-name is the single character with ASCII code <**asc_int**>.

46 cl

Usage: (cl <**clause**>)
(cl <**clause**> <**var**>)
(cl <**clause**> <**int1**> <**int2**>)
(cl <**clause**> <**int1**> <**var**>)
(cl <**clause**> <**var1**> <**int1**> <**int2**>)
(cl <**clause**> <**var1**> <**int**> <**var2**>)
"cl" succeeds if the clause pattern <**clause**> matches a standard clause, a menu clause, a dialog clause, a dempster clause, or a tuple of a base relation in the knowledge base. The predicate of <**clause**> must be instantiated to a constant. If more than one match is found for <**clause**>, alternatives are given on backtracking.
The second form of "cl" also finds the support associated with the matching clause or tuple, and binds it to <**var**>.
The three argument form of "cl" starts the search for a match to <**clause**> at the <**int1**>'th clause in the procedure. If <**clause**> matches the Nth clause, N is unified with the third argument. Thus a particular clause can be selected (third argument is an integer), or a clause numbering can be generated (third argument uninstantiated).
The four argument form is similar to three argument form, but the pair of support pairs associated with the clause is found and bound to <**var1**>.
Tuples of relations are held in an internally determined order, and the three and four argument indexing forms of "cl" will generate an error message if used on relations.

47 clear_current_window

Usage: (clear_current_window)
"clear_current_window" removes all text from the current Fril window. The background colour remains the same, and subsequent text added to the window

will appear in the most recently defined style.

48 close

Usage: (close <**filename**>)
"close" closes the input or output stream associated with the file <**filename**>. This stream must have been opened using "open", "create", or a related predicate. If the stream is open for output, then it is flushed before being closed.

49 close_dialog

Usage: (close_dialog <**const**>)
"close_dialog" dismisses the (modeless) dialog named by <**const**>. The dialog window is closed and removed from the screen, although the dialog itself is retained within the system and can be redisplayed using the "pose_dialog" built-in predicate.

50 complement

Usage: (complement <**named_fset**> <**var**>)
(complement <**named_fset**> <**var**> <**const**>)
Finds the complement of a fuzzy set, with respect to its universe. The two argument form returns an unnamed fuzzy set with no associated universe. The three argument form creates a new named fuzzy set with name <**const**> as a subset of the same universe as <**named_fset**>.

51 con

Usage: (con <**term**>)
"con" tests whether its argument is a constant.

52 confirm

Usage: (confirm <**const**>)
"confirm" displays a simple dialog box with a question and "Yes" and "No" buttons. <**const**> is the text of the question displayed in the dialog box.

53 conj

Usage: (conj <**atom1**> ... <**atomn**>)
"conj" implements the limiting case of the theoretical support logic programming calculus for conjunction, which is appropriate when the dependencies of associated propositions are unknown. It accumulates the support for successive pairs of the support logic goals <**atom1**> to <**atomn**> combining them using the limiting case calculus for conjunction described in Section »6.19. If the support pair for <**atom1**> is (n1 p1) and the support pair for <**atom2**> is (n2 p2), then the support pair for the goal (conj <**atom1**> <**atom2**>) is given by the following:
(MAX0, n1+n2-1, MINp1, p2)
This is in contrast to the default calculus which computes
(n1*n2, p1*p2).

54 convert

Usage: (convert <**term1**> <**term2**>)
"convert" checks that <**term1**> is the constant corresponding to the number <**term2**>, or performs the conversion between them.

55 cos

Usage: (cos <**term1**> <**term2**>)
"cos" checks that <**term2**> is the cosine of <**term1**>, which is assumed to be an angle in radians, or performs the cosine or arccosine functional conversion between them.

56 cprintf

Usage: (cprintf <**out_stream**> <**const**> <**term1**> ... <**termn**>)
"cprintf" performs formatted printing of terms <**term1**> to <**termn**> on stream <**out_stream**> in a format specified by <**const**>. This predicate corresponds very closely to the 'C' programming language print function 'fprintf', thus:
<**const**> is a control string which can contain ordinary characters and conversion specifications. Ordinary characters are simply written to <**out_stream**>. The conversion specifications refer to the successive arguments <**term1**> to <**termn**>, in order, and begin with the character "
(i) A minus sign indicating that the converted argument should be left adjusted within the field. The default is right adjustment.

(ii) A non-negative integer specifying a minimum field width. The converted number will be written in a field at least this wide. If the converted argument has fewer characters than the field width it will be appropriately padded on left or right, depending on whether the left adjustment indicator has been given. The padding character is a space, or zero if the field width was specified with a leading zero.
(iii) A full stop, which precedes the field width indicator.
(iv) A non-negative integer specifying the maximum number of characters to be written from a string, or the number of digits to be written to the right of the decimal point of a number. This is the precision. The conversion characters, which terminate the conversion specifications, characterise their arguments as follows:
c Argument is an ASCII code, and the corresponding character is written.
d Argument is an integer and is converted to fall in the range -32768 to +32767. Preceding the d by an "l", extends to "long" integer range -2147483648 to +2147483647.
e Argument is a floating point number and is converted to the following notation: (-)m.nnnnnnE(+ or -)xx where the length of the string of n's is specified by the precision. The default precision is 6.
f Argument is a floating point number and is converted to the following decimal notation: (-)mmm.nnnn where the length of string of n's is specified by the precision. The default is 6.
g Argument is a real number and is written using conversions "e" or "f", whichever is the shorter.
s Argument is a constant and is either written whole, or until the field width characters are exhausted.

57 cputime

Usage: (cputime <**var1**> <**var2**>)
"cputime" returns the current system time and the elapsed time since the last call to cputime. The time is, of course, only as accurate as the system clock.

58 create

Usage: (create <**filename**>)
"create" opens a file for output, and associates a stream with <**filename**>. If the file exists, it is overwritten.

59 create_a

Usage: (create_a <**filename**>)
"create_a" opens a file for output, and associates a stream with <**filename**>. If the file exists, further output is appended to the existing contents of the file. Either input or output operations may be used on the stream; however, sequences of input operations must be separated form sequences of output operations by a call to "flush" or "filepos".

60 create_r

Usage: (create_r <**filename**>)
"create_r" opens a file for output and/or input, and associates a stream with <**filename**>. If the file exists it is overwritten. The file can be read from or written to using the Fril input/output built-in predicates; however, when switching between reading and writing there must be an intervening call to "flush" or "filepos".

61 create_ra

Usage: (create_ra <**filename**>)
"create_ra" opens a file for output or input, and associates a stream with <**filename**>. If the file exists, further output is appended to the existing contents of the file. Either input or output operations may be used on the stream; however, sequences of input operations must be separated from sequences of output operations by a call to "flush" or "filepos".

62 create_window

Usage: (create_window <**const**> <**nni1**> <**nni2**> <**nni3**> <**nni4**> <**rgb1**> <**rgb2**>)
"create_window" creates a new window with the name specified in the first argument. The second two arguments <**nni1**> and <**nni2**> (both non-negative integers), specify the horizontal and vertical co-ordinates (in pixels) of the window on the screen. The fourth and fifth arguments, <**nni3**> and <**nni4**>, specify the horizontal and vertical size (again in pixels) of the window (including scroll bars etc.). The sixth and seventh arguments, <**rgb1**> and <**rgb2**>, are used to define the pen colour and background colour respectively, according to the codes given in Section 2.1

63 cscanf

Usage: (cscanf <**in_stream**> <**const**> <**var1**> ... <**varn**>)
"cscanf" performs formatted reading of terms from stream <**in_stream**> in a format specified by <**const**>. The terms are bound to variables <**var1**> to <**varn**>. This predicate corresponds very closely to the 'C' programming language input function 'fscanf', thus:
<**const**> is a control string which contains formats for the successive arguments <**var1**> to <**varn**> (except as detailed in (ii) below). Characters are read from the input stream up to a maximum format width (see (i) below), or until a character is found which does not conform to the format; this character will be read by the next read operation on the stream. The characters read in are converted into the appropriate type. Each format begins with the character
"
(i) A non-negative integer specifying a maximum field width, i.e. the maximum number of characters that can be read. If the first character read does not conform to the format, the variable is left unbound; otherwise it is bound to the term read from the stream. Note that with
(ii) An asterisk, indicating that the converted argument should be ignored. Input takes place as in (i) above, but no variable binding is made.
The conversion characters, which terminate the format specifications, characterise their arguments as follows:
c Reads a single character (or sequence of characters if a field width is given). The corresponding Fril variable is bound to a constant. This is the only conversion specification which reads all characters specified by the field width (unless the input is terminated by end of file).
s Reads a character string, terminated by a white space character or the field width. The corresponding Fril variable is bound to a constant. This specification differs from
d Reads a decimal integer, i.e. an optional sign and a sequence of digits terminated by field width or any character outside the range 0-9.
x Reads a hex integer, i.e. an optional sign and a sequence of digits and upper/lower case a-f, terminated by field width or any character outside the ranges 0-9, a-f.
o Reads an octal integer, i.e. an optional sign and a sequence of digits terminated by field width or any character outside the range 0-7.
e,f,g Reads a floating point number i.e. an optionally signed string of digits, possibly containing a decimal point, optionally followed by an exponent field, containing the letter 'e' or 'E', an optional sign, and an integer.

64 current_edit

Usage: (current_edit <**term**>)
"current_edit" checks whether <**term**> names the topmost edit window. If

<**term**> is a variable then "current_edit" will bind it to the name of the topmost edit window (if one exists). If <**term**> is a constant, the edit window of that name will be made the top window and will be brought to the top of the stack of windows on the screen.

65 current_window

Usage: (current_window <**term**>)
"current_window" checks whether <**term**> names the topmost Fril window. If <**term**> is a variable then "current_window" will bind it to the name of the topmost Fril window. If <**term**> is a constant, the Fril window of that name will be made the top window and will be brought to the top of the stack of windows on the screen.

66 cursor_position

Usage: (cursor_position <**term1**> <**term2**>)
"cursor_position" reports the current cursor position or moves the cursor to the column, row co-ordinates specified by <**term1**> and <**term2**> respectively.

67 date

Usage: (date <**var1**> <**var2**>)
"date" binds variables <**var1**> and <**var2**> to strings representing the date and time respectively. These strings are of the following form:
'Day Month Date Year' /* the date */
and 'HH:MM:SS' /* the time */

68 ddict

Usage: (ddict <**term**>)
"ddict" is used to check whether <**term**> is a named dtype fuzzy set, or to generate names of dtype fuzzy sets defined in the knowledge base.

69 def

Usage: (def <**term**>)
(def <**atom**>)

If <**term**> is a variable or a constant, (def <**term**>) is equivalent to (dict <**term**>) and checks whether <**term**> names a procedure in the knowledge base. If the argument to def is an <**atom**> representing a possible goal, def succeeds only if <**atom**> can be unified with some procedure in the knowledge base, otherwise def fails.

70 default_format

Usage: (default_format <**term1**> <**term2**>)
This built-in predicate enables the default text format for new windows to be reported or set under program control, using operations which correspond to those on the menu bar. <**term1**> can be a variable or a constant, and specifies the aspect of the text format to be considered i.e. style, size, font, justify, width. If <**term1**> is a variable, it will be successively instantiated to each of these on backtracking. <**term2**> can be a variable or one of the valid settings for the option <**term1**>.
When <**term1**> is one of size, font, justify, or width, <**term2**> can be either a variable or a constant. In the former case, the current setting is bound to <**term2**>, and in the latter case if <**term2**> names a valid setting then subsequent output will be in the new format.
If <**term1**> is style, <**term2**> can be a variable, in which case it is bound to a list of the currently active style settings. Alternatively, <**term2**> can be a list of options or a single option, which is the same as selecting those options from the style menu. Invalid options are ignored.

71 def_dtype

Usage: (def_dtype <**const**> <**dtype**>)
(def_dtype <**const**> <**dtype**> <**univ**>)
"def_dtype" defines <**const**> to be a named dtype with the associated fuzzy set <**dtype**>. If the three argument form is used, the fuzzy set is defined to be a subset of the universe <**univ**>.

72 def_itype

Usage: (def_itype <**const**> <**itype**>)
(def_itype <**const**> <**itype**> <**univ**>)
"def_itype" defines <**const**> to be a named itype with the associated fuzzy set <**itype**>. If the three argument form is used, the fuzzy set is defined to be a subset of the universe <**univ**>.

73 def_rel

Usage: (def_rel <**const**> <**nni**>)
"def_rel" defines <**const**> to be a relation of arity <**nni**>. Subsequent calls to addcl will add tuples to the relation <**const**>.

74 delcl

Usage: (delcl <**clause**>)
(delcl <**const**> <**int**>)
"delcl" deletes clauses from the knowledge base. The first form deletes the first clause which matches <**clause**>. The second form deletes the <**int**>th clause in the definition of the predicate named <**const**>.

75 delete_window

Usage: (delete_window <**const**>)
"delete_window" will delete the window named by <**const**>. If the window being deleted is the current window, then the window below on the stack will become the current window, otherwise the current window will remain the same.

76 dempdict

Usage: (dempdict <**term**>)
"dempdict" checks whether <**term**> names a dempster procedure in the knowledge base. If <**term**> is a variable, "dempdict" binds it to the name of a dempster procedure, assuming at least one dempster procedure has been defined. Different dempster procedure names can be generated by backtracking to "dempdict".

77 dempster

Usage: (dempster <**const**>)
"dempster" declares that support logic predicate name <**const**> is of type dempster, so that different clauses for the predicate correspond to independent viewpoints, and the supports are combined using the dempster combination rule.

78 dialog

Usage: (dialog (<**const1**> <**const2**> <**nni1**> <**nni2**> <**nni3**> <**nni4**> textbfdialog_items>)
"dialog" invokes the dialog compiler to create a modal or modeless dialog with name <**const1**>. The second argument <**const2**> specifies the type of the dialog, and must be "modal" or "modeless". The x and y co-ordinates of the top left hand corner of the dialog are given by <**nni1**> and <**nni2**> respectively; the horizontal and vertical dimensions of the dialog window are given by <**nni3**> and <**nni4**> respectively. The remaining terms specify the controls which appear on the dialog, and have the form specified in Chapter 4.
If <**const2**> is "modeless", subsequent clauses can be defined for <**const1**>, and goals of the form (<**const1**>are executed by clicking the controls of the dialog.

79 dict

Usage: (dict <**term**>)
"dict" checks whether <**term**> names a procedure in the knowledge base. If <**term**> is a variable, dict binds it to the name of a procedure, and can be resatisfied by binding <**term**> to different procedure names.

80 disable_dlog_item

Usage: (disable_dlog_item <**const1**> <**const2**> <**const3**>)
(disable_dlog_item <**const1**> <**const2**> <**const3**> <**const4**>)
Disables an item on a user-defined dialog (i.e. the item is displayed in grey, and does not respond to mouse clicks). The type of control is given by <**const1**>, and can be one of static, edit, cluster, popup, button, check_box (3 argument form) or popup, radio_button (4 argument form).
<**const2**> is the dialog name, <**const3**> is the control, and (optionally) <**const4**> is a component of the control, i.e. the name of a radio button in the cluster, or an entry on the popup menu.

81 disable_menu

Usage: (disable_menu <**const**>)
(disable_menu <**const1**> <**const2**>)
The one-argument form of disable_menu disables the user-defined menu named by the predicate <**const**>, so that the title on the menu bar and all options are displayed in grey, and cannot be selected using the mouse. The two-argument

form disables the single item <**const2**> on the user-defined menu named by the predicate <**const1**>.

82 disj

Usage: (disj <**atom1**> ... <**atomn**>)
"disj" implements the limiting case of the theoretical support logic programming calculus for disjunction, which is appropriate when the dependencies of associated propositions are unknown. It accumulates the support for successive pairs of the support logic goals <**atom1**> to <**atomn**> combining them using the limiting case calculus for disjunction described in Section »6.19». If the support pair for <**atom1**> is (n1 p1) and the support pair for <**atom2**> is (n2 p2), then the support pair for the goal (disj <**atom1**> <**atom2**>) is given by the following:
(MAXn1, n2, MIN1, p1+p2)
This is in contrast to the default calculus which uses "or" instead of "disj" and computes the support pair (n1+n2-n1*n2, p1+p2-p1*p2)

83 division

Usage: (division <**num1**> <**num2**> <**term**>)
"division" is used to unify <**term**> with the integer value produced by dividing <**num1**> by <**num2**>. If <**term**> is a variable, then it will be bound to that value. This is equivalent to performing the real division and truncating the result. The goal is not resatisfiable.

84 dlogdict

Usage: (dlogdict <**term**>)
"dlogdict" is used to check whether <**term**> names a user-defined dialog, or to generate names of user-defined dialogs in the knowledge base.

85 dtype

Usage: (dtype <**term**>)
"dtype" checks whether <**term**> is a discrete fuzzy set (named or otherwise)

86 dtype_name

Usage: (dtype_name <**var**> <**named_dtype**>)
(dtype_name <**const**> <**var**>)
(dtype_name <**const**> <**named_dtype**>)
"dtype_name" converts a constant into the dtype definition named by that constant, and vice-versa. It can also be used to check that a constant names a dtype. The first form binds <**var**> to the name of the dtype <**named_dtype**>. The second form binds <**var**> to the definition of the dtype named by <**const**>. The third form checks that <**const**> is the name of <**named_dtype**>.

87 dtype_to_freq

Usage: (dtype_to_freq <**var**> <**list**>)
(dtype_to_freq <**dtype-fuzzy-set**> <**term**>)
NOT YET IMPLEMENTED

] for i=2,, then <**var**> will be bound to a dtype fuzzy set containing the vertices specified by ni with memberships given by mi . In the second form, a list of element]**dtype_to_freq converts between a dtype fuzzy set and a list of element-frequency pairs representing the least prejudiced distribution. In the first form of usage, if <list> is bound to a list of k lists, each of the form (ni mi) such that ni > ni+1 for i=2and mi is in the range [0,1] for i=2,, then <var> will be bound to a dtype fuzzy set containing the vertices specified by ni with memberships given by mi . In the second form, a list of element**/frequency pairs is created from the least prejudiced distributuoin corresponding to the dtype fuzzy set, and this list is unified with <**term**>

88 dtype_to_list

Usage: (dtype_to_list <**var**> <**list**>)
(dtype_to_list <**dtype**> <**term**>)

] then <**var**> will be bound to a dtype fuzzy set containing the elements specified by the constants, with memberships given by the numbers. In the second form, a list of element]**"dtype_to_list" converts between a discrete fuzzy set and a list of element-membership pairs. In the first form of usage, if <list> is bound to a list of lists, each containing a constant and a number in the interval [0,1] then <var> will be bound to a dtype fuzzy set containing the elements specified by the constants, with memberships given by the numbers. In the second form, a list of element**/membership pairs is created from the discrete fuzzy set <**dtype**>,

and this list is unified with <**term**>.

89 dynamic

Usage: (dynamic <**const**>)
(dynamic (<**const1**> ... <**constn**>))
This predicate is only relevant in the context of the construction of a module. "dynamic" is used to declare as dynamic, predicate name <**const**> in the first form, or each of the predicates <**const1**> to <**constn**> in the second form. This is necessary when they (or it) are to be examined or altered at run time, by some other part of the same module.

90 edict

Usage: (edict <**term**>)
"edict" checks whether <**term**> names an export procedure. Thus, "edict" contains a dictionary of all the predicates exported by currently loaded modules, i.e. all module-defined user callable predicates.

91 edit

Usage: (edit <**const**>)
This built-in predicate enables the operations on the edit menu to be performed under program control. The operations affect the current selection or cursor location in the topmost window, irrespective of its type. <**const**> should be one of
cut deletes the current selection and copies it to the clipboard
copy copies the current selection to the clipboard
paste copies the contents of the clipboard to the current selection or cursor position
clear deletes the current selection

92 editdict

Usage: (editdict <**term**>)
"editdict" checks to see whether there is an edit window currently open with the name <**term**>. If <**term**> is a constant, the goal succeeds only if there is an edit window with that name. If <**term**> is a variable, it will be bound to the name of an existing edit window. In this case, the goal is resatisfiable, and

will successively return the names of edit windows in the order that they were opened. If no edit windows are open, the goal fails.

93 enable_dlog_item

Usage: (enable_dlog_item <**const1**> <**const2**> <**const3**>)
(enable_dlog_item <**const1**> <**const2**> <**const3**> <**const4**>)
"enable_dlog_item" is used to enable an item on a user-defined dialog (i.e. the item is displayed in its normal colour, and responds to mouse clicks when appropriate). The type of control is given by <**const1**>, and can be one of static, edit, cluster, popup, button, check_box (3 argument form) or popup, radio_button (4 argument form).
<**const2**> is the dialog name, <**const3**> is the control, and (optionally) <**const4**> is a component of the control, i.e. the name of a radio button in the cluster, or an entry on the popup menu.

94 enable_menu

Usage: (enable_menu <**const**>)
(enable_menu <**const1**> <**const2**>)
The one-argument form of "enable_menu" forces the user-defined menu named by the predicate <**const**> to be enabled, so that the title on the menu bar and all options are displayed in black, and can be selected using the mouse from Fril top level. The two-argument form enables the single item <**const2**> on the user-defined menu named by the predicate <**const1**>.

95 eq

Usage: (eq <**term1**> <**term2**>)
"eq" attempts to unify its two arguments. If it is embedded in a support query, the model of semantic unification specified by the supp_query goal is used.

96 errm

Usage: (errm <**const**>)
(errm <**var**>)
The first form of "errm" alters the error message status to <**const**>. When an error occurs, the current error message status, <**const**>, is passed to the error handler and can be used to control the action on the occurrence of errors.

When the default error handler "error" is being used, the following options are expected:
"y" This enables all error messages to be written to the stream "stderr". The appropriate error action is performed.
"n" This turns off all control error messages that do not have the action "abort", and all interface error messages. The appropriate error action is still performed. Input, support and module errors are always reported by the default error handler. When the second form of "errm" is used, the variable becomes bound to the current error message status. The default status is "y"

97 error

Usage: (error <**int1**> control <**list**> <**const1**> <**atom**> <**const2**>)
(error <**int1**> input <**list**> <**const1**> <**const2**> <**int2**>)
(error <**int1**> support <**list**> <**const1**> <**atom**> <**var**>)
(error <**int1**> module <**list**> <**const1**> <**const2**> <**const3**>)
(error <**int1**> interface <**list**> <**const1**> <**atom**> <**const2**>)
(error <**int1**> dialog <**list**> <**const1**> <**atom**> <**const2**>)
Prints error number <**int1**> and error message contained in <**list**>, according to the type of error and the current error message status, <**const1**>. The remaining arguments are dependent on the type of error, and contain further information about the source of the error, and the appropriate action to take after the message is printed. See Section »>16.25 for a complete description of these arguments.

98 err_han

Usage: (err_han <**const**>)
(err_han <**var**>)
"err_han" reports or determines which error handler is to be used to handle errors reported by the system. At start-up this defaults to the system error handler "error". Error handling can be redirected to a user defined procedure named <**const**>. The procedure with this predicate name will then be used to define the consequent behaviour.

99 evlog

Usage: (evlog (<**itype**> <**atom1**> <**wt1**> **atomn**> <**wtn**>))
(evlog (<**atom1**> <**wt1**> **atomn**> <**wtn**>))
"evlog" implements the Fril evidential support logic combination of goals in the body. It evaluates a weighted sum of necessary and possible supports, which

are then scaled according to the fuzzy set <**itype**>.

100 exists

Usage: (exists <**const**>)
"exists" checks whether <**const**> is the name of an existing file. The ".frl" extension is not assumed and it must be explicitly in the string <**const**> where necessary. If no path is given, then "exists" looks in the current directory. If a path is given, "exists" searches the specified path for the file.

101 exit

Usage: (exit <**int**>)
"exit" terminates execution of Fril, and returns integer <**int**> to the parent process (if appropriate to the operating system).

102 exp

Usage: (exp <**term1**> <**term2**>)
"exp" checks that <**term2**> is the natural exponential of <**term1**>, i.e. <**term2**> is "e" to the power of <**term1**>, or alternatively <**term1**> is the natural logarithm of <**term2**>.

103 expected_value

Usage: (expected_value ???)
???

104 export

Usage: (export <**const**>)
(export (<**const1**> ... <**constn**>))
This predicate is only relevant in the context of the construction of a module. In the first form, "export" is used to declare predicate name <**const**> as export. In the second form, each of the predicates <**const1**> to <**constn**> is declared as export. Export declarations are necessary for all predicates defined in a module which are to be callable by the user. All predicates so declared can

then be called from Fril top-level, or from other modules.

105 fail

Usage: (fail)
"fail" is the Fril Prolog "negation as failure" primitive. In Fril prolog (depth search) execution mode, the goal (fail) can never be satisfied, and it causes Fril to backtrack. If fail is encountered when the system is evaluating supports, an error message occurs.

106 fdict

Usage: (fdict <**term**>)
"fdict" checks whether <**term**> names a foreign language procedure. Thus, "fdict" contains a dictionary of all the predicates made accessible to Fril in the compilation of the foreign language interface routines using "xfril".

107 file

Usage: Usage: (file <**const1**> <**const2**>)
This built-in predicate enables the operations on the file menu to be performed under program control. <**const2**> specifies a filename or the name of an edit window. <**const1**> specifies the operation to be performed, which should be one of
new <**const2**> is a filename. An edit window is created and the file <**const2**> is overwritten or created.
open <**const2**> is the name of an existing file. An edit window is created for this file.
save <**const2**> is the name of an edit window whose contents are written to file.
close <**const2**> is the name of an edit window, which is closed after saving the contents.

108 filebcopy

Usage: (filebcopy <**in_stream**> <**out_stream**> <**int**>)
(filebcopy <**in_stream**> <**out_stream**> <**char**>)
"filebcopy" copies a block of bytes from <**in_stream**> to <**out_stream**>. The starting position of the block is the current position in <**in_stream**>, and

the size of the block is defined by the last argument. The first form copies a block of size <**int**> bytes, and the second form copies up to and including the first occurrence of the character <**char**> in the <**in_stream**>. The resultant position in the <**in_stream**> will, in each case be the byte immediately following the last byte to be copied.

109 filename

Usage: (filename <**const**> <**term1**> <**term2**> <**term3**>)
filename <**var**> <**const1**> <**const2**> <**const3**>)
"filename" is used to convert between a fully specified file name and its associated path, root and extension. In the first form, <**const**> is a fully specified filename, and the goal will attempt to unify the path with <**term1**>, the root with <**term2**> and the extension with <**term3**>. If any of <**term1**>, <**term2**> and <**term3**> are variables, they are bound to the appropriate components of the complete filename <**const**>, if possible. In the second form, <**var**> is bound to the full filename constructed from the path <**const1**>, the root <**const2**> and the extension <**const3**>.

110 filepos

Usage: (filepos <**filename**> <**var**>)
(filepos <**filename**> <**nni**>)
"filepos" binds <**var**> to the current offset in file <**filename**>, or changes the offset to <**nni**>.

111 filesearch

Usage: (filesearch <**filename**> <**const**>)
"filesearch" searches forwards in the file <**filename**> from the current position until it finds <**const**>, which marks the new offset.

112 findall

Usage: (findall <**term**> <**atom_list**> <**var**>)
"findall" finds all solutions to the list of goals <**atom_list**>, and for each solution collects <**term**> appropriately instantiated into a list which is unified with <**var**>.

113 find_file

Usage: (find_file <**const**>)
"find_file" puts up a standard file dialog but includes only files named <**const**> and directories in the display. The user can navigate around directories in the usual manner.

114 flush

Usage: (flush <**in_stream**>)
(flush <**out_stream**>)
(flush <**io_stream**>)
"flush" clears the buffer associated with an input or output stream. If the stream is open for output, all buffered data is written to the associated file. Using flush on an input stream causes all immediately available input to be discarded.

115 forall

Usage: (forall <**atom_list1**> <**atom_list2**>)
"forall" repeatedly executes <**atom_list1**> and, for every solution and associated variable bindings, <**atom_list2**> is executed.

116 format

Usage: (format <**term1**> <**term2**>)
This built-in predicate enables the current text format to be reported or set under program control, using operations which correspond to those on the menu bar. <**term1**> can be a variable or a constant, and specifies the aspect of the text format to be considered i.e. style, size, font, justify, width. If <**term1**> is a variable, it can be successively instantiated to each of these on backtracking. <**term2**> can be a variable or one of the valid settings for the option <**term1**>.
When <**term1**> is one of size, font, justify, or width, <**term2**> can be either a variable or a constant. In the former case, the current setting is bound to <**term2**>, and in the latter case if <**term2**> names one of the menu items subsequent output will be in the new format.
If <**term1**> is style, <**term2**> can be a variable, in which case it is bound to a list of the currently active style settings. Alternatively, <**term2**> can be a list of options or a single option, which is the same as selecting those options from the style menu. Invalid options are ignored.

117 fset

Usage: (fset <**term**>)
"fset " checks whether <**term**> is a universe, a continuous (itype) or a discrete (dtype) fuzzy set (named or otherwise)

118 fsetdict

Usage: (fsetdict <**term**>)
"fsetdict" is used to check whether <**term**> is a universe or named fuzzy set, or to generate names of universes or fuzzy sets defined in the knowledge base. Both itype (continuous) and dtype (discrete) fuzzy sets are included.

119 fset_name

Usage: (fset_name <**var**> <**named_fset**>)
(fset_name <**const**> <**var**>)
(fset_name <**const**> <**named_fset**>)
"fset_name" converts a constant into the fuzzy set or universe definition named by that constant, and vice-versa. It can also be used to check that a constant names a fuzzy set or universe. The first form binds <**var**> to the name of the fuzzy set <**named_fset**>. The second form binds <**var**> to the definition of the fuzzy set or universe named by <**const**>. The third form checks that <**const**> is the name of <**named_fset**>.

120 gen

Usage: (gen <**const**> (<**atom_list1**> **atom_listn**>) (<**support1**>textbfsupportn>))
"gen" implements a generalised version of the extended Fril rule, where <**const**> specifies the support logic operator to be used in combining the supports for goals in each <**atom_list**> (e.g. "conj" or "and"). Each <**atom_list**> is a conjunction of support logic goals, and the list <**atom_list1**> **atom_listn**> represents a mutually exclusive and exhaustive set of conditions. Each <**support**> is a conditional support pair for the corresponding <**atom_list**>.

121 general

Usage: (general (<**atom_list1**> **atom_listn**>) (<**support1**>textbfsupportn>))
"general" implements the extended Fril rule. Each <**atom_list**> is a conjunc-

tion of support logic goals, and the list <**atom_list1**> **atom_listn**> represents a mutually exclusive and exhaustive set of conditions. Each <**support**> is a conditional support pair for the corresponding <**atom_list**>.

122 gensym

Usage: (gensym <**const**> <**var**>)
"gensym" generates a new symbol or string from a root specified in <**const**>. <**var**> becomes instantiated to a string which comprises the characters of <**const**> followed by one or more digits. Each time Fril is started up, the first call to (gensym <**root**> <**var**>) instantiates <**var**> to the string <**root**> followed by index 1. On each subsequent call of gensym, with the same <**root**>, the index is incremented by 1 for that <**root**> only.

123 get

Usage: (get <**in_stream**> <**var**>)
"get" reads the next non-space character from the input stream <**in_stream**>, and binds its ASCII code to <**var**>. Space and control characters are ignored.

124 getb

Usage: (getb <**in_stream**> <**var**>)
"getb" reads the next character from the input stream <**in_stream**>, and binds its ASCII code to <**var**>. All characters, including space and controls are returned.

125 getenv

Usage: (getenv <**const**> <**term**>)
"getenv" checks that the environment variable (for example "frlib") given by <**const**> has the value given by <**term**>. The values of environment variables "frlib", "frpath" and "frhelp" are typically set to appropriate directory paths.

126 getk

Usage: (getk <**var**>)
"getk" reads the next character made available from the keyboard and binds

its ASCII code to <**var**>. All characters, including space and controls are returned. This goal will succeed as soon as the character is typed and does not have to wait for a carriage return to put the character on the standard input stream (stdin). Note that the <**carriage-return**> or <**Enter**> (ASCII code 10) typed at the end of a top level query is always left in the standard input stream, and if this is not the character that is required, then the stream should first be flushed using the goal (flush stdin).

127 get_directory

Usage: (get_directory <**var**>)
Displays a standard file dialog containing only directories. The user can navigate around directories in the usual manner. If the call succeeds, <**var**> is bound to the name of the chosen directory on completion.

128 get_dlog_val

Usage: (get_dlog_val <**const1**> <**const2**> <**const3**> <**var**>)
Reports the value of a control on a user-defined dialog. The type of control is given by <**const1**>, and can be one of static, edit, cluster, popup, button, or check_box.
<**const2**> is the dialog name, <**const3**> is the control name, and <**var**> is bound to the current value:
Control Value
static text displayed
button text displayed
edit text in edit box
cluster name of selected radio button
check_box 'on' or 'off'
popup menu item selected

129 get_filename

Usage: (get_filename <**var**>)
"get_filename" displays a standard file selection dialog box and binds the name of the file selected to <**var**>.

130 get_prior

Usage: (get_prior <**univ**> <**var**>)
(get_prior <**univ**> <**term**>)
"get_prior" extracts the prior probability distribution associated with the universe <**univ**>.

131 get_univ

Usage: (get_univ <**named_fset**> <**var**>)
If a named fuzzy set is defined as a subset of a universe, get_univ gives access to the universe from the fuzzy set definition.

132 help

Usage: (help <**const**>)
"help" displays help on a built-in predicate or a topic named by <**const**>. For a built-in predicate, this help consists of a short description of the behaviour of the predicate and the arguments it requires. The topics covered are mainly the sections of chapter 6, e.g. manipulating windows, modules, system limits etc., and the help provided is a summary of the relevant facilities or information. The goal (help help) will display a list of the topics for which help is currently provided.

133 help_example

Usage: (help_example <**const**>)
"help_example" displays an example of the use of the built-in predicate <**const**>.

134 hide_window

Usage: (hide_window <**const**>)
"hide_window" will put the window named by <**const**> at the bottom of the stack of windows on the screen. If the window to be hidden is the current window, then the window next on the stack will become the current window, otherwise the current window will remain the same.

135 highlight

Usage: (highlight <**int1**> <**int2**>)
(highlight <**var1**> <**var2**>)
"highlight" affects the top window and reports or establishes the current selection. In the first form, highlight selects the characters in the range <**int1**> to <**int2**>, where <**int2**> **int1**>. Values are truncated if either is outside the range 0 - number of characters in the window.
In the second form, <**var1**> and <**var2**> are bound to the start and end of the current selection (note that these may be the same).

136 idict

Usage: (idict <**term**>)
"idict" is used to check whether <**term**> is a named itype fuzzy set, or to generate the names of itype fuzzy sets defined in the knowledge base.

137 if

Usage: (if <**atom**> <**atom_list1**> <**atom_list2**>)
"if" simulates the if ... then ... else control construct of conventional procedural languages.

138 import

Usage: (import <**const**>)
(import (<**const1**> ... <**constn**>))
This predicate is only relevant in the context of the construction of a module. In the first form, "import" is used to declare predicate name <**const**> as import. In the second form, each of the predicates <**const1**> to <**constn**> is declared as import. This is necessary for all predicates for which there are no definitions in the module, but which are called by procedures in the module.

139 inform

Usage: (inform <**const**>)
"inform" displays a simple dialog box with a statement and an "OK" button. <**const**> is the text of the statement displayed in the dialog box.

140 init_random

Usage: (init_random <**num**>)
Seeds the random number generator with the value <**num**>. Different seeds will cause different pseudo-random sequences to be generated by the built-in predicate "random".

141 int

Usage: (int <**term**>)
Checks whether <**term**> is an integer.

142 intersection

Usage: (intersection <**fset1**> <**fset2**> <**var**>)
(intersection <**fset1**> <**fset2**> <**fset3**>)
"intersection" generates or checks the intersection between two fuzzy sets. In the first form, <**var**> is bound to the intersection of the two fuzzy sets <**fset1**> and <**fset2**>; the second form is used to check that the result of intersecting fuzzy sets <**fset1**> and <**fset2**> is <**fset3**>.

143 intok

Usage: (intok <**in_stream**> <**var**>)
"intok" reads the next language token from the input stream <**in_stream**>. Tokens are constants, numbers and variables, and non-blank separators such as brackets etc. This means that lists and itypes will not be read as single items, but each bracket and each element will be read separately.

144 isall

Usage: (isall <**var**> <**term**> <**atom1**> ... <**atomn**>)
i.e. (isall <**var**> <**term**>|<**atom_list**>)
"isall" finds all solutions to the series of goals <**atom1**>, ... <**atomn**>, and for each solution collects <**term**> appropriately instantiated to a list which is unified with <**var**>.

145 itype

Usage: (itype <**term**>)
Checks whether <**term**> is an itype (named or otherwise).

146 itype_name

Usage: (itype_name <**var**> <**named_itype**>)
(itype_name <**const**> <**var**>)
(itype_name <**const**> <**named_itype**>)
"itype_name" converts a constant into the itype definition named by that constant, and vice-versa. It can also be used to check that a constant names an itype. The first form binds <**var**> to the name of the itype <**named_itype**>. The second form binds <**var**> to the definition of the itype named by <**const**>. The third form checks that <**const**> is the name of <**named_itype**>.

147 itype_to_freq

Usage: (itype_to_freq <**var**> <**list**>)
(itype_to_freq <**itype-fuzzy-set**> <**term**>)
NOT YET IMPLEMENTED

] for i=2,, then <**var**> will be bound to an itype fuzzy set containing the vertices specified by ni with memberships given by mi . In the second form, a list of element]**itype_to_freq converts between an itype fuzzy set and a list of element-frequency pairs representing the least prejudiced distribution. In the first form of usage, if <list> is bound to a list of k lists, each of the form (ni mi) such that ni > ni+1 for i=2and mi is in the range [0,1] for i=2,, then <var> will be bound to an itype fuzzy set containing the vertices specified by ni with memberships given by mi . In the second form, a list of element**/frequency pairs is created from the least prejudiced distributuoin corresponding to the itype fuzzy set, and this list is unified with <**term**>

148 itype_to_list

Usage: (itype_to_list <**var**> <**list**>)
(itype_to_list <**itype-fuzzy-set**> <**term**>)

] for i=2,, then <**var**> will be bound to an itype fuzzy set containing the vertices specified by ni with memberships given by mi . In the second form, a

list of element]**itype_to_list converts between an itype fuzzy set and a list of element-membership pairs. In the first form of usage, if <list> is bound to a list of k lists, each of the form (ni mi) such that ni > ni+1 for i=2and mi is in the range [0,1] for i=2,, then <var> will be bound to an itype fuzzy set containing the vertices specified by ni with memberships given by mi . In the second form, a list of element**/membership pairs is created from the itype fuzzy set, and this list is unified with <**term**>

149 kb_garbage

Usage: (kb_garbage)
"kb_garbage" reclaims space that has been released by deleting items from the knowledge base.

150 kill

Usage: (kill <**const**>)
(kill <**named_fset**>)
"kill" deletes an entire procedure definition, a fuzzy set definition or a module. The first form deletes the procedure or module named <**const**>. The second form deletes the definition of the fuzzy set <**named_fset**>.

151 less

Usage: (less <**num1**> <**num2**>)
(less <**const1**> <**const2**>)
"less" checks the ordering of terms, either numerically (when both arguments are numbers), or lexicographically (when both arguments are constants).

152 less_eq

Usage: (less_eq <**num1**> <**num2**>)
(less_eq <**const1**> <**const2**>)
"less_eq" checks the ordering of terms, either numerically (when both arguments are numbers), or lexicographically (when both arguments are constants).

153 list

Usage: (list all)
(list <**var**>)
(list <**const**>)
(list <**named_fset**>)
"list" displays the definition of a procedure or named fuzzy set defined in the knowledge base, at the standard output. If the argument to list is the symbol "all" or an uninstantiated variable, the entire knowledge base is displayed. Otherwise, if the argument is a constant symbol, which names the procedure or fuzzy set, this is displayed. "list" does not display built-in predicates, or procedures that are defined in modules.

154 listfile

Usage: (listfile <**filename**> all)
(listfile <**filename**> <**var**>)
(listfile <**filename**> <**const**>)
(listfile <**filename**> <**named_fset**>)
"listfile" displays the definition of a procedure or named fuzzy set defined in the knowledge base, to <**filename**>. If the second argument to listfile is the symbol "all" or an uninstantiated variable, the entire knowledge base is displayed. Otherwise, if it is a constant symbol, which names the procedure or fuzzy set, this is displayed. "listfile" does not display built-in predicates, or procedures that are defined in modules.

155 lload

Usage: (lload (<**filename1**> ... <**filenamen**>))
"lload" recursively loads each of the source code or module files, named in its single list argument. Thus lload loads files in the order <**filename1**>, <**filename2**>, ... , <**filenamen**>.

156 load

Usage: (load <**filename**>)
"load" loads into Fril, the data in file <**filename**>. This data can take two forms: (i) valid Fril input, i.e. knowledge base definitions and queries, or (ii) Compiled modules. In the first case, "load" redirects Fril current input so that the system takes input from the file <**filename**> until end_of_file is read. At this point the current input is reset to its prior stream. When "loading" source

files, any procedures defined in the file will be ADDED to procedures of the same name, that are already defined in the knowledge base. Other name clashes will generate error messages. In the second case, "load" brings a compiled module (with the ".frm" extension) into system memory, thereby allowing access, via the exported predicates, to all the procedures defined in the module. The module name is entered into the module dictionary, accessed by "mdict", and the export and visible predicates are entered into the export dictionary, accessed by "edict". If the module name, or the name of any of the export or visible predicates, is the same as any knowledge base predicate or named itype, loaded module, or predicate in the export dictionary, then the module will not load.

157 lreload

Usage: (lreload (<**filename1**> ... <**filenamen**>))
"lreload" recursively reloads each of the source code or module files named in its single list argument. Thus lreload reloads files in the order <**filename1**>, <**filename2**>, ... , <**filenamen**>.

158 match

Usage: (match <**term1**> <**term2**>)
"match" is only used in a support evaluation context, and is the semantic unification counterpart to the built-in predicate "eq". If either <**term1**> , <**term2**> or both is a fuzzy set, then "match" will succeed with support calculated as the conditional support for <**term1**> given <**term2**>. The order of terms is important, since the support for <**term1**> given <**term2**> will usually differ from the support for <**term2**> given <**term1**>. This contrasts with the syntactic unification of "eq" which is symmetric. If neither term is a fuzzy set definition, match succeeds with support (1 1) if its arguments unify, and support (0 0) otherwise.

159 maximise_frame

Usage: (maximise_frame)
This predicate is only applicable to Windows 3.1 systems, where all Fril and edit windows are treated as offspring of the parent window and (for example) cannot move outside the display area defined by the parent window. "maximise_frame" expands the parent window to fill the whole screen, and allows maximum display area for Fril and edit windows. This is equivalent to clicking the "maximise" button at the top right of the window.

160 max_window_pos

Usage: (max_window_pos <**var1**> <**var2**>)
"max_window_pos" reports the maximum x and y co-ordinates that a window can be moved to, whilst still remaining partially on the screen.

161 max_window_size

Usage: (max_window_size<**var1**> <**var2**>)
"max_window_size" reports the maximum x and y size that a window can be expanded to, whilst still remaining completely on the screen.

162 mdict

Usage: (mdict <**term**>)
"mdict" checks whether <**term**> names a module which is currently loaded into the Fril system. Thus, "mdict" contains a dictionary of all the currently loaded modules.

163 menu

Usage: (menu <**const**>)
"menu" is used to declare that a menu named <**const**> should be added to the menu bar. The declaration should appear before any clauses are defined for the predicate <**const**>. Each clause should have exactly one constant argument, which is added to the menu named <**const**>. If the user selects an item (say <**const2**>) from the menu <**const**>, then the Fril goal (<**const**> <**const2**> is executed.

164 menudict

Usage: (menudict<**term**>)
"menudict" checks whether <**term**> names a procedure in the knowledge base that has been declared to be of type "menu". If <**term**> is a variable, menudict binds it to the name of a menu procedure, and can be resatisfied by binding <**term**> to different menu procedure names.

165 min_window_pos

Usage: (min_window_pos <**var1**> <**var2**>)
"min_window_pos" reports the minimum x and y co-ordinates that a window can be moved to, whilst still remaining on the screen.

166 min_window_size

Usage: (min_window_size <**var1**> <**var2**>)
"min_window_size" reports the minimum x and y size of a Fril or edit window.

167 module

Usage: (module <**filename1**> <**filename2**>)
"module" takes a file <**filename1**> of Fril program text, crafted into module form using "export", "import", "visible" and "dynamic" declarations as appropriate, and creates a new module file <**filename2**>, which is given the extension ".frm".

168 module_initialisation

Usage: (module_initialisation <**atom_list**>)
"module_initialisation" is a keyword recognised by the module compiler, and used to set up goals for execution when a module is loaded. In normal execution, module_initialisation is exactly equivalent to the query '?'. If it appears in a file which is compiled into a module, the goals are added to a list of goals executed after the module code has been loaded. It can be used to ensure other modules are loaded, or perform initialisation of dynamic data. Note that any predicates called by a compiled "module_initialisation" goal must be either built-in predicates, or defined as import, export, visible, or dynamic within the module.

169 move_dialog

Usage: (move_dialog <**const**> <**term1**> <**term2**>)
"move_dialog" reports or establishes the position of the top left hand corner, of the dialog window named <**const**>, as the x and y co-ordinates specified by <**term1**> and <**term2**> respectively.

170 move_window

Usage: (move_window <**const**> <**term1**> <**term2**>)
"move_window" reports or establishes the position of the top left hand corner, of the window named <**const**>, as the column, row co-ordinates specified by <**term1**> and <**term2**> respectively.

171 name

Usage: (name <**var**> <**const**>)
(name (<**asc_int1**> <**asc_int2**> ...) <**var**>) (name (<**asc_int1**> <**asc_int2**> ...) <**const**>)
"name" converts a list of ASCII integers into the constant whose print name is that series of ASCII codes, or vice- versa. Alternatively, name can be used to check that a constant is named by the string of ASCII codes given.

172 neg

Usage: (neg <**atom**>)
"neg" implements the standard Fril prolog "negation as failure". The <**atom**> must be a callable Fril prolog goal. "neg" is similar to "negg", except that the former takes a single argument goal.

173 negg

Usage: (negg <**const**> <**term1**> ... <**termn**>)
i.e. (negg|<**atom**>)
"negg" implements the standard Fril prolog "negation as failure". The <**atom**> must be a callable Fril prolog goal. "negg" has identical semantics to neg, but a slightly different syntax. "negg" is used by inserting the symbol "negg" at the front of the list representing the goal <**atom**> being negated. This syntax is compatible with "not" (used for support logic negation).

174 nospy

Usage: (nospy all)
(nospy <**const**>)
"nospy all", in the first form of usage, removes all spy points from predicates

currently being spied. In the second form of usage, "nospy" removes the spy-point from the procedure name <**const**>.

175 not

Usage: (not <**const**> <**term1**> .... <**termn**>)
i.e. (not|<**atom**>)
"not" is the support logic negation operator, so that if the support for the conclusion (<**const**> <**term1**>...<**termn**>) is (L U), the negated call has support (1-U 1-L).

176 num

Usage: (num <**term**>)
Checks whether <**term**> is a number (i.e. floating point decimal or integer).

177 oh

Usage: (oh <**atom_list**>)
"oh" executes the series of calls in <**atom_list**> using the standard depth-first mechanism, and prints the successful instantiation of <**atom_list**> on the standard output. If the query evaluation succeeds, "oh" prompts the user to respond and backtracks to find another solution if requested, otherwise it simply succeeds.

178 open

Usage: (open <**filename**>)
"open" opens a file for input, and associates a stream with <**filename**>.

179 or

Usage: (or <**atom1**> ... <**atomn**>)
"or" implements a Fril support logic disjunction. It accumulates the support for successive pairs of the support logic goals <**atom1**> to <**atomn**> combining them using the support logic calculus for disjunction.

180 orr

Usage: (orr <**atom_list1**> ... <**atom_listn**>)
"orr" implements a standard Fril prolog disjunction using the depth search inference strategy. To prove the goal, the <**atom_list**>'s are successively queried starting with <**atom_list1**> and continuing to <**atom_listn**> until one of them is proved. If none can be proved then the call to "orr" fails.

181 os

Usage: (os <**atom_list**>)
(os (<**const**> <**atom1**> **atomn**>))
i.e. (os (<**const**> | <**atom_list**>))
"os" finds the support for the conjunction of goals in <**atom_list**>, one solution at a time. "os" uses supp_query for support evaluation, so that for each possible instantiation of <**atom_list**>, all proof paths are evaluated and the supports combined. The second form of the query allows the semantic unification method to be specified for the query evaluation - <**const**> can be "match" (the default), "point_match", or "poss_match". "os" prints the successful instantiation of <**atom_list**> together with its computed support, at the standard output. After each solution is printed, "os" prompts the user to respond and backtracks to find another solution if requested, otherwise it succeeds printing a completion message.

182 osc

Usage: (osc <**atom_list**>)
(osc (<**const**> <**atom1**> **atomn**>))
i.e. (osc (<**const**> | <**atom_list**>))
"osc" finds the support for the conjunction of goals in <**atom_list**>, one solution at a time. "osc" uses the built-in predicate "supp_collect" for support evaluation, so that for each possible instantiation of <**atom_list**>, all proof paths are evaluated and the supports combined. Any fuzzy sets in the same argument position in <**atom_list**> are combined into a single expected fuzzy set and defuzzified. The second form of the query allows the semantic unification method to be specified for the query evaluation - <**const**> can be "match" (the default), "point_match", or "poss_match". "osc" prints the successful instantiation of <**atom_list**> together with its computed support, at the standard output. After each solution is printed, "osc" prompts the user to respond and backtracks to find another solution if requested, otherwise it succeeds printing a completion message.

183 ose

Usage: (ose <**atom_list**>)
(ose (<**const**> <**atom1**> **atomn**>))
i.e. (ose (<**const**> | <**atom_list**>))
"ose" finds the support for the conjunction of goals in <**atom_list**>, one solution at a time. "ose" uses the built-in predicate "supp_expect" for support evaluation, so that for each possible instantiation of <**atom_list**>, all proof paths are evaluated and the supports combined. Any fuzzy sets in the same argument position in <**atom_list**> are combined into a single expected fuzzy set. The second form of the query allows the semantic unification method to be specified for the query evaluation - <**const**> can be "match" (the default), "point_match", or "poss_match". "ose" prints the successful instantiation of <**atom_list**> together with its computed support, at the standard output. After each solution is printed, "ose" prompts the user to respond and backtracks to find another solution if requested, otherwise it succeeds printing a completion message.

184 osv

Usage: (osv <**atom_list**>)
(osv (<**const**> <**atom1**> **atomn**>))
i.e. (osv (<**const**> | <**atom_list**>))
"osv" finds the support for the conjunction of goals in <**atom_list**>, one solution at a time. "osv" uses the built-in predicate "supp_value" for support evaluation, so that for each possible instantiation of <**atom_list**>, all proof paths are evaluated and the supports combined. Any fuzzy sets in the same argument position in <**atom_list**> are combined into a single expected fuzzy set and defuzzified. The second form of the query allows the semantic unification method to be specified for the query evaluation - <**const**> can be "match" (the default), "point_match", or "poss_match". "osv" prints the successful instantiation of <**atom_list**> together with its computed support, at the standard output. After each solution is printed, "osv" prompts the user to respond and backtracks to find another solution if requested, otherwise it succeeds printing a completion message.

185 p

Usage: (p <**term1**> <**term2**> .... <**termn**>)
"p" prints its arguments on the standard output, separated by spaces. The next print operation commences immediately after the last character printed. (p) on its own prints a single space. Strings are printed without quotes, so that any control characters within a string may affect the display. Named fuzzy sets

appear as definitions.

186 peek

Usage: (peek <**in_stream**> <**var**>)
"peek" examines the next non-space printing character in the input stream <**in_stream**>, and binds its ASCII code to <**var**>. Space and control characters are ignored. The character is NOT read, and is available to the next input operation on the stream.

187 peekb

Usage: (peekb <**in_stream**> <**var**>)
"peekb" examines the next character (including space and control characters) in the input stream <**in_stream**>, and binds its ASCII code to <**var**>. The character is NOT read, and it is available to the next input operation on the stream.

188 peekk

Usage: (peekk <**var**>)
"peekk" examines the next character (including space and control characters) made available from the keyboard and binds its ASCII code to <**var**>. The character is NOT read, and is available to the next input operation on the standard input stream (stdin). Note that the <**carriage- return**> or <**Enter**> (usually ASCII code 10) typed at the end of a top level query is always left in the standard input stream and if this is not the character that is required, then the stream should first be flushed using the goal (flush stdin).

189 pen_col

Usage: (pen_col <**RGB**>)
(pen_col <**var**>)
(pen_col <**nni**>)
"pen_col" reports or establishes the local pen colour of the current Fril window. In the first form, the pen colour is changed to that represented by the code <**RGB**> as defined in Section 2.1. Any characters subsequently printed to the window will be printed using this new colour. A pen colour is associated only with the topmost Fril window at the time the goal is called, and will not

be affected by going to another window and changing the pen colour for that window. In the second form, <**var**> is bound to a list of three integers representing the pen colour of the current Fril window.

190 point_match

Usage: (point_match <**term1**> <**term2**>)
"point_match" is only used in a support evaluation context, and is a specialisation of the semantic unification mechanism. If both <**term1**> and <**term2**> are fuzzy sets, then "point_match" will succeed with support calculated as the point value conditional support for <**term1**> given <**term2**>. The order of terms is important, since the support for <**term1**> given <**term2**> will differ, in general, from the support for <**term2**> given <**term1**>. This contrasts with the syntactic unification of "eq" which is symmetric. If neither term is a fuzzy set definition, point_match succeeds with support (1 1) if its arguments unify, and support (0 0) otherwise.

191 pose_dialog

Usage: (pose_dialog <**const**> <**var**>)
"pose_dialog" displays the dialog window named by <**const**>. If the dialog is modal, execution is suspended until the dialog is dismissed by clicking on one of its controls. In this case, <**var**> is bound to the name of the control which dismissed the dialog. Otherwise, if the dialog is modeless, <**var**> is bound to the constant "modeless" and execution continues.

192 poss_match

Usage: (poss_match <**term1**> <**term2**>)
"poss_match " is only used in a support evaluation context, and implements the possibilistic version of semantic unification. If either <**term1**> , <**term2**> or both is a fuzzy set, then "poss_match " will succeed with support calculated as the possibilistic conditional support for <**term1**> given <**term2**>. The order of terms is important, since the support for <**term1**> given <**term2**> will differ, in general, from the support for <**term2**> given <**term1**>. This contrasts with the syntactic unification of "eq" which is symmetric. If neither term is an fuzzy set definition, match succeeds with support (1 1) if its arguments unify, and support (0 0) otherwise.

193 power

Usage: (power <**num1**> <**num2**> <**var**>)
(power <**num1**> <**var**> <**num2**>)
"power" is used to raise a number to the power of its exponent, or to compute the logarithm of a number with specified base. In the first form, <**num1**> is raised to the power of <**num2**> to produce a real result in <**var**>. In the second form, the logarithm of <**num2**> to base <**num1**> is computed to produce a real result in <**var**>.

194 pp

Usage: (pp <**term1**> <**term2**> .... <**termn**>)
"pp" prints each term followed by a new line on the standard output. (pp) on its own prints a new line. Strings are printed without quotes, so that any control characters within a string may affect the display. Named fuzzy sets appear as their itype or dtype definitions.

195 ppq

Usage: (ppq <**term1**> <**term2**> .... <**termn**>)
"ppq" prints each term followed by a new line on the standard output. (ppq) on its own prints a new line. A string is printed in quotes if it contains any characters that would cause it to be classified as something other than a constant, such as spaces or control characters. The name of a fuzzy set is printed, rather than its definition.

196 pq

Usage: (pq <**term1**> <**term2**> .... <**termn**>)
"pq" prints each term followed by a space on the standard output. (pq) on its own prints a space. A string is printed in quotes if it contains any characters that would cause it to be classified as something other than a constant, such as spaces or control characters. The name of a fuzzy set is printed, rather than its definition.

197 prlen

Usage: (prlen <**term**> <**var**>)
(prlen <**term**> <**nni**>)

"prlen" reports or checks the print length of <**term**> which can be any term apart from a list, i.e. constant, number (real or integer), variable or fuzzy set. The print length of <**term**> is the number of character locations that will be used when <**term**> is printed to a stream in unquoted form. For variables, this will be the length of the internal representation, e.g. _136 which has length 4. The length of fuzzy set definitions will also account for the spaces that would be printed by Fril. "prlen" cannot be used directly on a list - this must be recursively processed using "prlen" on each element. In the first form, "prlen" binds <**var**> to the non-negative integer which is the print length of <**term**>. In the second form, "prlen" checks that <**nni**> is the print length of <**term**>.

198 pspaces

Usage: (pspaces <**nni**>)
"pspaces" prints <**nni**> spaces on the standard output (screen), where <**nni**> is a non-negative integer.

199 putb

Usage: (putb <**out_stream**> <**asc_int**>)
"putb" prints the character whose ASCII code is <**asc_int**> on the output stream associated with <**out_stream**>.

200 qh

Usage: (qh <**atom_list**>)
"qh" executes the series of calls in <**atom_list**> using the standard depth-first search mechanism, and prints the successful instantiations of <**atom_list**> on the standard output. "qh" backtracks to find all possible solutions without any further interaction from the user, and it finally succeeds printing a completion message whether or not any solutions were found.

201 qs

Usage: (qs <**atom_list**>)
(qs (<**const**> <**atom1**> **atomn**>))
i.e. (qs (<**const**> | <**atom_list**>))
"qs" finds the support for the conjunction of goals in <**atom_list**> for all possible instances of <**atom_list**>. "qs" uses supp_query for support evaluation,

so that for each possible instantiation of <**atom_list**>, all proof paths are evaluated and the supports combined. The second form of the query allows the semantic unification method to be specified for the query evaluation - this can be "match" (the default), "point_match", or "poss_match". "qs" backtracks to find all possible solutions without any further interaction from the user, and for each solution prints the instantiation of <**atom_list**> together with its computed support pair, on the standard output. "qs" finally succeeds printing a completion message.

202 qsc

Usage: (qsc <**atom_list**>)
(qsc (<**const**> <**atom1**> **atomn**>))
i.e. (qsc(<**const**> | <**atom_list**>))
"qsc" finds the support for the conjunction of goals in <**atom_list**> for all possible instances of <**atom_list**>. "qsc" uses supp_collect for support evaluation, so that for each possible instantiation of <**atom_list**>, all proof paths are evaluated and the supports combined. Any fuzzy sets in the same argument position in <**atom_list**> are combined into a single expected fuzzy set. The second form of the query allows the semantic unification method to be specified for the query evaluation - this can be "match" (the default), "point_match", or "poss_match". "qsc" backtracks to find all possible solutions without any further interaction from the user, and for each solution prints the instantiation of <**atom_list**> together with its computed support pair, on the standard output. "qsc" finally succeeds printing a completion message.

203 qse

Usage: (qse <**atom_list**>)
(qse (<**const**> <**atom1**> **atomn**>))
i.e. (qse(<**const**> | <**atom_list**>))
"qse" finds the support for the conjunction of goals in <**atom_list**> for all possible instances of <**atom_list**>. "qse" uses supp_expect for support evaluation, so that for each possible instantiation of <**atom_list**>, all proof paths are evaluated and the supports combined. Any fuzzy sets in the same argument position in <**atom_list**> are combined into a single expected fuzzy set. The second form of the query allows the semantic unification method to be specified for the query evaluation - this can be "match" (the default), "point_match", or "poss_match". "qse" backtracks to find all possible solutions without any further interaction from the user, and for each solution prints the instantiation of <**atom_list**> together with its computed support pair, on the standard output. "qs" finally succeeds printing a completion message.

204 qsv

Usage: (qsv<**atom_list**>)
(qsv (<**const**> <**atom1**> **atomn**>))
i.e. (qsv (<**const**> | <**atom_list**>))
"qsv" finds the support for the conjunction of goals in <**atom_list**> for all possible instances of <**atom_list**>. "qsv" uses supp_defuz for support evaluation, so that for each possible instantiation of <**atom_list**>, all proof paths are evaluated and the supports combined. Any fuzzy sets in the same argument position in <**atom_list**> are combined into a single expected fuzzy set and defuzzified. The second form of the query allows the semantic unification method to be specified for the query evaluation - this can be "match" (the default), "point_match", or "poss_match". "qsv" backtracks to find all possible solutions without any further interaction from the user, and for each solution prints the instantiation of <**atom_list**> together with its computed support pair, on the standard output. "qsv" finally succeeds printing a completion message.

205 r

Usage: (r <**var**>)
"r" reads the next term from the current Fril input stream, and binds it to <**var**>.

206 random

Usage: (random <**var**>)
"random" generates pseudorandom floating point numbers in the range 0 to 1.

207 rdict

Usage: (rdict <**term**>)
"rdict" generates or checks relation names.

208 read

Usage: (read <**in_stream**> <**var**>)
"read" reads the next term from the input stream <**in_stream**>, and binds it

to <**var**>.

209 read_suppterm

Usage: (read_suppterm <**in_stream**> <**var1**> <**var2**>)
"read_suppterm" reads the next term and a support from the input stream <**in_stream**>, and binds the term to <**var1**> and the support to <**var2**>. If an <**end-of-line**> character is read before a support is found, <**var2**> is bound to ((1 1)(0 1)).

210 reload

Usage: (reload <**filename**>)
"reload" transfers into Fril, the data in file <**filename**>. This data can take two forms: (i) Fril valid input, i.e. knowledge base definitions and queries, or (ii) Compiled modules. In the first case, "reload" redirects Fril current input so that the system takes input from the file <**filename**> until end_of_file is read. At this point the current input is reset to its prior stream. When "reloading" source files, any procedures and named fuzzy sets defined in the file will REPLACE items of the same name that have already been defined in Fril (i.e. procedures, named fuzzy sets, modules and predicates in the export dictionary), and the new definitions will be stored in the knowledge base. In the second case, "reload" brings a compiled module (with the ".frm" extension) into system memory, thereby allowing access, via the export predicates, to all the procedures defined in the module. The module name is entered into the module dictionary, accessed by "mdict", and the export and visible predicates are entered into the export dictionary, accessed by "edict". Any existing knowledge base procedures, named fuzzy sets, loaded modules or predicates in the export dictionary, which have the same name as the new module or any predicates which it exports or makes visible, will be replaced.

211 remainder

Usage: (remainder <**num1**> <**num2**> <**term**>)

]**"remainder" checks or computes <term> as the remainder on dividing <num1> by <num2>. If <term> is a variable, then it will be bound to that value. <term> should be equal to <num1> - <num2>*trunc(<num1>/<num2**>), where "trunc" denotes rounding down to the nearest integer.

212 repeat

Usage: (repeat)
"repeat" provides a simple mechanism for iteration through backtracking.

213 resize_window

Usage: (resize_window <**const**> <**term1**> <**term2**>)
"resize_window" checks or establishes the size of the window named <**const**> measured by its width and depth (in pixels) specified by <**term1**> and <**term2**> respectively. The position of a window is defined by the co-ordinates of its top left-hand corner, and so the process of resizing a window is carried out relative to its stationary top left-hand corner.

214 sc

Usage: (sc <**atom**>)
(sc <**atom**> <**const**>)
"sc" is a support logic meta-predicate which succeeds with the support for goal <**atom**>, executed using the supp_collect querying mechanism. Solutions containing fuzzy sets are combiend using the expected fuzzy set »> BWP

215 scroll_top_window

Usage: (scroll_top_window <**const1**> <**const2**>)
scroll_top_window" scrolls the text display in the top window in the direction <**const1**>, by an amount <**const2**>. Acceptable combinations of direction and amount are
Direction Amount
up, down page, line
left, right page, character
The movement of text in the window is the same as that produced by clicking the scroll bar or its arrows.

216 se

Usage: (se <**atom**>)
(se <**atom**> <**const**>)
"se" is a support logic meta-predicate which succeeds with the support for goal

<**atom**>, executed using the supp_expect querying mechanism. Solutions containing fuzzy sets are combiend using the expected fuzzy set »> BWP

217 set

Usage: set (<**const**> <**num1**> <**num2**>))
set (<**const**> (<**elem1**> <**elem2**> **elemn**>)) where <**elemi**> is a constant or number and <**num1**> is less than <**num2**>
(set <**const**> neg_inf <**num2**>)
(set <**const**> <**num2**> pos_inf)
"set" defines a universe for fuzzy sets. The universe may be a continuous interval (first case shown above) or a discrete set (second case). Open intervals may be represented using the third and fourth cases, where pos_inf and neg_inf refer to positive infinity and negative infinity respectively.

218 setenv

Usage: (setenv <**const1**> <**const2**>)
"setenv" is used to set the value of the environment variable <**const1**> to <**const2**>. The values of environment variables can be retrieved using "getenv".

219 set_difference

Usage: (set_difference<**fset1**> <**fset2**> <**var**>)
(set_difference<**fset1**> <**fset2**> <**fset3**>)
"set_difference" generates or checks the set difference of two fuzzy sets. In the first form, <**var**> is bound to the set difference of the two fuzzy sets <**fset1**> and <**fset2**>; the second form is used to check that the result of taking the set difference of fuzzy sets <**fset1**> and <**fset2**> is <**fset3**>.

220 set_dlog_val

Usage: (set_dlog_val <**const1**> <**const2**> <**const3**> <**const4**>)
(set_dlog_val popup <**const2**> <**const3**> <**list**>)
Sets the value of a control on a user-defined dialog. The type of control is given by <**const1**>, and can be one of static, edit, cluster, popup, button, or check_box.
<**const2**> is the dialog name, <**const3**> is the control name, and <**const4**> is the new value for the control:

Control Value
static text displayed
button text displayed
edit text in edit box
cluster name of selected radio button
check_box 'on' or 'off'
popup menu item selected.
In the second form, "set_dlog_val" can be used to replace the list of items on a popup menu. <**const2**> is the dialog name, <**const3**> is the popup name, and <**list**> is a list of constants which replace the items on the popup. If <**list**> is the empty list, the popup is disabled.

221 sh

Usage: (sh <**const**>)
"sh" passes its argument to the operating system as a command to be executed.

222 sin

Usage: (sin <**term1**> <**term2**>)
"sin" checks that <**term2**> is the sine of <**term1**>, which is assumed to be an angle in radians, or performs the sine or arcsine functional conversion between them.

223 snips

Usage: (snips <**const**> <**term1**> ... <**termn**>)
i.e. (snips|<**atom**>)
"snips" makes its goal argument deterministic. The meaning of this is as follows: whether or not <**atom**> is resatisfiable, (snips|<**atom**>) is not resatisfiable.

224 spy

Usage: (spy <**const**>)
"spy" sets a spy point on procedure named <**const**>. This is used in association with the leap option to give the user control over trace interaction and trace display output in the Fril prolog trace package.

225 spying

Usage: (spying <**term**>)
"spying" checks whether <**term**> names a spied procedure. This is only useful in the context of the Fril prolog trace package. If <**term**> is a variable, "spying" binds it to the name of a procedure and can be resatisfied by binding <**term**> to different spied procedure names.

226 square

Usage: (square <**term1**> <**term2**>)
"square" checks that <**term2**> is the square of <**term1**>, where both arguments are numbers, or performs the square or positive square root functional conversion between them.

227 stack_windows

Usage: (stack_windows)
"stack_windows" resizes all Fril and edit windows to a standard size and cascades them on the screen starting at the top left hand corner and moving towards the bottom right. Fril windows appear in front of edit windows, and older windows appear in front of newer windows. Thus stdwnd is always placed at the front after this built-in predicate is executed.

228 statistics

Usage: (statistics)
"statistics" prints the current space used by each of the five stacks, the knowledge base, and the symbol table. It also prints the time used by the process.

229 stricteq

Usage: (stricteq <**term1**> <**term2**>)

]**"stricteq" checks for strict equality of two terms. No unification is carried out, so that if <term1> is a variable, stricteq succeeds only if <term2> has previously been unified with <term1>. If <term1> and <term2> are lists, then stricteq succeeds if corresponding elements are strictly equal. Fuzzy sets are considered to be strictly equal**

if they contain strictly equal element/membership pairs.

230 stringof

Usage: (stringof <**list**> <**var**>)
(stringof <**var**> <**const**>)
(stringof <**list**> <**const**>)
In the first form of usage, if <**list**> is bound to a list of constants, each of which is a single character, then <**var**> will be unified with the corresponding string of those characters. In the second form of usage, if <**const**> is bound to a string, then <**var**> is unified with the corresponding list of its individual character constants. In the third form of its usage, both arguments are instantiated.

231 str_to_list

Usage: (str_to_list <**const**> <**list**>)
(str_to_list <**const**> <**var**>)
(str_to_list <**var**> <**list**>)
The built-in predicate "str_to_list" allows conversion between strings and lists of tokens. In the first two forms the string <**const**> is broken down into a list of input tokens which is unified with the second argument. Alternatively, in the third usage, the print names of the list elements are concatenated and the result is unified with <**var**>.

232 sum

Usage: (sum <**term1**> <**term2**> <**term3**>)
"sum" is used to perform addition and subtraction, and succeeds if <**term1**>+<**term2**> = <**term3**>. If two of the arguments are numbers and the third is a variable, the latter is instantiated so that the above relation holds. If the arguments are an itype, a number, and a variable, or two simple itypes and a variable, the variable is instantiated using fuzzy arithmetic. Simple itypes are normalised trapezoidal or triangular itypes, i.e. of the form [a:0 b:1 c:1 d:0] or [a:0 b:1 c:0]

233 supp_collect

Usage:
»> BWP

234 supp_expect

Usage:
»> BWP

235 supp_query

Usage: (supp_query <**atom_list**> <**var_support**>)
(supp_query <**atom_list**> <**var_support**> <**const**>)
"supp_query" finds the support for each solution to the goal <**atom_list**>, binding it to <**var_support**>. The second form of the query allows the semantic unification method to be specified for the query evaluation - this can be "match" (the default), "point_match", or "poss_match". Execution is in breadth-search mode, so that all proof paths are evaluated and the supports combined. It is important that the goal being solved should have a finite search tree, and that all solutions to the goal should be ground (i.e. they should not contain variables or fuzzy sets). This is most easily accomplished by ensuring that all facts used in finding a solution are ground, and that there are no anonymous variables in rules.

236 supp_value

Usage:
»> BWP

237 sv

Usage:
»> BWP

238 sys

Usage: (sys <**const**>)
"sys" checks whether <**const**> is the name of a built-in predicate.

239 system_garbage

Usage: (system_garbage)
"system_garbage" reclaims space that has been used by temporary system items created in the knowledge base. This occurs during execution of the built-in predicates "supp_expect", "supp_value", "qse", "qsv", "ose", "osv", "wse", "wsv", and "gensym". "system_garbage" kills intermediate predicates, then calls "kb_garbage" to reclaim the space.

240 tan

Usage: (tan <**term1**> <**term2**>)
"tan" checks that <**term2**> is the tangent of <**term1**>, which is assumed to be an angle in radians, or performs the tangent or arctangent functional conversion between them.

241 tempfile

Usage: (tempfile <**var**>)
"tempfile" generates a unique temporary filename of the form frlXXXXX, where XXXXX is a system-dependent string of characters. If the environment variable FRTEMP is specified, the filename is prefixed with the directory named by FRTEMP.

242 times

Usage: (times <**term1**> <**term2**> <**term3**>)
"times" is used to perform multiplication and division, and succeeds if <**term1**>*<**term2**> = <**term3**>. If two of the arguments are numbers and the third is a variable, the latter is instantiated so that the above relation holds. If the arguments are an itype, a number, and a variable, or two simple itypes and a variable, the variable is instantiated using fuzzy arithmetic. Simple itypes are normalised trapezoidal or triangular itypes, i.e. of the form [a:0 b:1 c:1 d:0] or [a:0 b:1 c:0]

243 top_window

Usage: (top_window <**term**>)
"top_window" checks whether <**term**> names the currently active window, which can be either a Fril window or an edit window. If <**term**> is a variable,

then it is bound to the name of the active window. If <**term**> is a constant and names a Fril window or an edit window, then that window will be made the active window and brought to the top of the stack of windows on the screen.

244 tq

Usage: (tq <**atom_list**>)
(tq <**const**>)
In the first form, "tq" invokes the Fril prolog tracer on the conjunction of goals in <**atom_list**>. "tq" mimics the built-in predicate "?", following exactly the same depth first execution strategy that Fril prolog uses, but it allows users to follow proof path inferences step by step, observing the instantiations of successful goals. The user can interact with the interpreter for goal execution and inference, and can control the amount of trace information that is displayed. In the second form, the user can change trace defaults with <**const**> set to "setup" or "reset".

245 true

Usage: (true <**support**>)
"true" acts as though it were a goal which succeeded with support pair <**support**>.

246 truncate

Usage: (truncate <**float**> <**int**>)
(truncate <**var**> <**int**>)
(truncate <**float**> <**var**>)
"truncate" converts integers to floating point numbers and vice-versa.

247 union

Usage: (union <**fset1**> <**fset2**> <**var**>)
(union <**fset1**> <**fset2**> <**fset3**>)
"union" generates or checks the union of two fuzzy sets. In the first form, <**var**> is bound to the union of the two fuzzy sets <**fset1**> and <**fset2**>; the second form is used to check that the result of forming the union of fuzzy sets <**fset1**> and <**fset2**> is <**fset3**>.

248 univ

Usage: (univ <**term**>)
"univ" checks whether <**term**> names a defined universe for fuzzy sets. The universe may be continuous or discrete.

249 univdict

Usage: (univdict <**term**>)
"univdict" is used to check whether <**term**> is a universe for fuzzy sets, or to generate names of universes defined in the knowledge base.

250 univ_name

Usage: (univ_name <**var**> <**univ**>)
(univ_name <**const**> <**var**>)
(univ_name <**const**> <**univ**>)
"univ_name" converts a constant into the universe named by that constant, and vice-versa. It can also be used to check that a constant names a given universe The first form binds <**var**> to the name of the universe <**univ**>. The second form binds <**var**> to the definition of the universe named by <**const**>. The third form checks that <**const**> is the name of the universe <**univ**>.

251 var

Usage: (var <**term**>)
"var" tests whether its argument is an uninstantiated variable.

252 visible

Usage: (visible <**const**>)
(visible (<**const1**> ... <**constn**>))
This predicate is only relevant in the context of the construction of a module. "visible" is used to declare as visible, predicate name <**const**> in the first form, or each of the predicates <**const1**> to <**constn**> in the second form. This is necessary when they (or it) are to be called by a meta-call from within the same module. Meta-calls will otherwise only be able to make calls to the user accessible predicates. In terms of user accessibility, visible predicates are bound by the same rules and behave in the same way as export predicates.

253 w

Usage: (w <**out_stream**> <**term1**> <**term2**> **termn**>)
"w" prints its arguments, separated by spaces, on the output stream <**out_stream**>. The next print operation on <**out_stream**> commences immediately after the last term printed. (w <**out_stream**>) with no other arguments prints a single space on the stream. Strings are printed without quotes, and fuzzy sets appear as itype or dtype definitions.

254 wdict

Usage: (wdict <**term**>)
"wdict" checks to see if there is a window currently defined with the name <**term**>. If <**term**> is a constant, then the goal only succeeds if there is a window with that name. If <**term**> is a variable, then it is bound to the name of an existing window. In this case, the goal is resatisfiable and <**term**> is successively bound to the names of windows in the order that they were created - therefore beginning with stdwnd.

255 wh

Usage: (wh (<**term**> <**atom1**> ... <**atomn**>))
i.e. (wh (<**term**>|<**atom_list**>))
"wh" executes the series of calls in <**atom_list**> using the standard depth-first search mechanism, and prints the corresponding instantiations of <**term**> for which <**atom_list**> succeeds, at the standard output. "wh" backtracks to find all possible solutions without any further interaction from the user and finally succeeds printing a completion message, whether or not any solutions were found.

256 wq

Usage: (wq <**out_stream**> <**term1**> <**term2**> .... <**termn**>)
"wq" prints its arguments, separated by spaces, on the output stream <**out_stream**>. The next print operation on <**out_stream**> commences immediately after the last term printed. (w <**out_stream**>) with no other arguments prints a single space on the stream. Strings are printed with quotes if necessary, and a named fuzzy set appears as the name, rather than the definition.

257 write

Usage: (write <**out_stream**> <**term1**> ... <**termn**>)
"write" prints each term followed by a new line on the output stream <**out_stream**>. The goal (write <**out_stream**>) with no other arguments prints a new line on the stream. Strings are printed without quotes, and fuzzy sets appear as itype or dtype definitions.

258 writeq

Usage: (writeq <**out_stream**> <**term1**> ... <**termn**>)
"writeq" prints each term followed by a new line on the output stream <**out_stream**>. Goal (writeq <**out_stream**>) with no other arguments prints a new line on the stream. Strings are printed with quotes if necessary, and a named fuzzy set appears as the name, rather than the definition.

259 ws

Usage: (ws (<**term**> <**atom1**> ... <**atomn**>))
i.e. (ws (<**term**>|<**atom_list**>))
(ws (<**const**> <**term**> <**atom1**> ... <**atomn**>))
"ws" finds the support for the conjunction of goals in <**atom_list**> for all possible instances of <**atom_list**>. "ws" uses "supp_query" for support evaluation so that for each possible instantiation of <**atom_list**> all proof paths are evaluated and the supports combined. The second form of the query allows the semantic unification method to be specified for the query evaluation - <**const**> can be "match" (the default), "point_match", or "poss_match". "ws" backtracks to find all possible solutions, and for each solution found, the corresponding instantiations of <**term**> together with the solution's computed support are printed at the standard output. "ws" finally succeeds printing a completion message.

260 wse

Usage: (wse (<**term**> <**atom1**> ... <**atomn**>))
i.e. (wse (<**term**>|<**atom_list**>))
(wse (<**const**> <**term**> <**atom1**> ... <**atomn**>))
"wse" finds the support for the conjunction of goals in <**atom_list**> for all possible instances of <**atom_list**>. "ws" uses "supp_expect" for support evaluation so that for each possible instantiation of <**atom_list**> all proof paths

are evaluated and the supports combined. Any fuzzy sets in the same argument position in <**atom1**> **atomn**> are combined into a single expected fuzzy set. The second form of the query allows the semantic unification method to be specified for the query evaluation - <**const**> can be "match" (the default), "point_match", or "poss_match". "wse" backtracks to find all possible solutions, and for each solution found, the corresponding instantiations of <**term**> together with the solution's computed support are printed at the standard output. "wse" finally succeeds printing a completion message.

261 wsc

Usage: (wsc (<**term**> <**atom1**> ... <**atomn**>))
i.e. (wsc (<**term**>|<**atom_list**>))
(wsc (<**const**> <**term**> <**atom1**> ... <**atomn**>))
"wsc" finds the support for the conjunction of goals in <**atom_list**> for all possible instances of <**atom_list**>. "wsc" uses "supp_value" for support evaluation so that for each possible instantiation of <**atom_list**> all proof paths are evaluated and the supports combined. Any fuzzy sets in the same argument position in <**atom1**> **atomn**> are combined into a single expected fuzzy set. The second form of the query allows the semantic unification method to be specified for the query evaluation - <**const**> can be "match" (the default), "point_match", or "poss_match". "wsc" backtracks to find all possible solutions, and for each solution found, the corresponding instantiations of <**term**> together with the solution's computed support are printed at the standard output. "wsc" finally succeeds printing a completion message.

262 wspaces

Usage: (wspaces <**out_stream**> <**nni**>)
"wspaces" prints <**nni**> spaces on the output stream <**out_stream**>, where <**nni**> is a non-negative integer.

263 wsv

Usage: (wsv(<**term**> <**atom1**> ... <**atomn**>))
i.e. (wsv(<**term**>|<**atom_list**>))
(wsv(<**const**> <**term**> <**atom1**> ... <**atomn**>))
"wsv" finds the support for the conjunction of goals in <**atom_list**> for all possible instances of <**atom_list**>. "wsv" uses the built-in predicate "supp_value" for support evaluation so that for each possible instantiation of <**atom_list**> all proof paths are evaluated and the supports combined. Any fuzzy sets in

the same argument position in <**atom1**> **atomn**> are combined into a single expected fuzzy set and then defuzzified to give a single value. The second form of the query allows the semantic unification method to be specified for the query evaluation - <**const**> can be "match" (the default), "point_match", or "poss_match". "wsv" backtracks to find all possible solutions, and for each solution found, the corresponding instantiations of <**term**> together with the solution's computed support are printed at the standard output. "wsv" finally succeeds printing a completion message.

264 zoom_top_window

Usage: (zoom_top_window <**const**>)
"zoom_top_window" allows the currently active window to be zoomed out to fill the screen, or revert to its original size (before zooming). The argument <**const**> can be "out", causing the window to fill the screen or "in", which causes the window to revert to its former size. These predicates give a similar effect to clicking in the zoom box.