Report 80-24
Stanford -- KSL

Scientific DataLink

The MRS Dictionary.
Michael R. Genesereth, Russell Greiner,
Milton R. Grinberg, David E. Smith,
Dec 1980

Stanford Heuristic Programming Project
Report No. HPP-80-24

December 1980
revised January 1984

The MRS Dictionary

Michael R. Genesereth
Russell Greiner
Milton R. Grinberg
David E. Smith

Heuristic Programming Project
Department of Computer Science
School of Humanities and Sciences
Stanford University

Table of Contents

1. Dictionary

***** (* <a_1> . . . <a_n> <a_{n+1}>)
means that <a_{n+1}> represents the product of the values represented by <a_1> . . . <a_n>. See **fq**.

+ (+ <a_1> . . . <a_n <a_{n+1}>)
means that <a_{n+1}> represents the sum of the values represented by <a_1> . . . <a_n>. See **fq**.

- (- <a> <b> <c>)
means that <c> represents the difference between <a> and <b>. See **fq**.

// (// <a> <b> <c>)
means that <c> represents the quotient of <a> and <b>. See **fq**.

< (< <a> <b>)
means that <a> is less than <b>. See **rq**.

<= (<= <a> <b>)
means that <a> is less than or equal to <b>. See **rq**.

= (= <a> <b>)
means that the terms <a> and <b> are synonymous, i.e. they refer to the same object. See **lookup-=**.

> (> <a> <b>)
means that <a> is greater than <b>. See **rq**.

>= (>= <a> <b>)
means that <a> is greater than or equal to <b>. See **rq**.

achieve (achieve <p>)
makes the proposition <p> true. **Achieve** is an abstract operator implemented using **kb** and **toachieve**. (**achieve** <p>) and (**achieve** (**not** <p>)) now work when the proposition <p> begins with **value**, **property**, **repn**, **thrrepn**, **includes**, **indb**, **better**, or **primitive**. So, up to a point, does (**achieve** (**if** <q> <p>)), which calls **trueps** on <q> and then **ACHIEVE**s <p> with the resulting bindings **plugged** in. Note that propositions stashed in theories other than the currently writeable one are not affected. (e.g. (**achieve** '(**repn** <p> <r>)) has the result that all propositions matching <p> will henceforth be **stashed**, **lookup**'ed, and so on using representation <r>. This includes re-storing currently accessible propositions that were stored using **pr-stash** or in **cnf**. **repn** works on all theories that are active when it is **achieved**. (**achieve** '(**threpn** <p> <r> <th>))) does the same thing as with **repn**, but affecting only theory <th>. It is the user's responsibility to ensure that there are never two or more theories active which use different representations for the same proposition.) See **kb**, **repn**, **threpn**

achieve-if **(achieve-if (if <p> <q>))**
has the effect of calling **achieve** on the proposition **<q>** for each list of variable bindings that makes proposition **<p>** true, with the relevant bindings substituted into **<q>**. See **achieve**, **trueps**, **plug**.

achieve-not **(achieve-not (not <p>))**
is an abstract operator implemented using **kb** and **tounachieve**. When called with an argument of the form **(not <p>)**, it is supposed to achieve the opposite of **<p>**, if meaningful. See **achieve**, **unachieve**.

achieve-repn **(achieve-repn (repn <prop> <rpn>))**
uses **repn-assert** to switch the representation of **<prop>** from its old value to **<rpn>**, and converts any instances of **<prop>** that can be found under the old representation to the new one. See **domain**, **repn**, **repn-assert**, **repn-method**.

achieve-threpn **(achieve-threpn (repn <prop> <rpn> <th>))**
is identical to **achieve-repn** except that it sets the currently writable theory to **<th>** temporarily while it executes. See **achieve-repn**, **theory**, **threpn**.

activate **(activate $\langle t_1 \rangle$ ... $\langle t_n \rangle$)**
makes the propositions in the theories $\langle t_1 \rangle$ ... $\langle t_n \rangle$ available for retrieval or deduction. See **theory**, **activetheories**, **deactivate**.

activetheories
has as its value the list of currently active theories. The propositions in these theories are available for retrieval by **pr-lookup** and **pr-lookups**. See **pr-stash**, **pr-unstash**, **pr-lookup**, **pr-lookups**, **activate**, and **deactivate**.

agenda **agenda**
is a list of **applicable** tasks. See **applicable**, **scheduler**.

and **(and $\langle p_1 \rangle$ . . . $\langle p_n \rangle$)**
means that the propositions $\langle p_1 \rangle$ . . . $\langle p_n \rangle$ are all true. See **assert-and**, **bc**, **br**, **fc**.

applicable **(applicable <k>)**
states that the task **<k>** is applicable and, therefore, executable unless it is disqualified. See **executable**, **disqualified**, **scheduler**.

arity **(arity <rel> <i>)**
provides typing information, indicating that the relation (or operation) **<rel>** takes **<i>** arguments, e.g. **(arity arity 2)**. See **domain**.

ask **(ask <p>)**
calls **output**, prints the result, and reads the user's answer. If **<p>** is a ground proposition, **ask** tries to obtain an answer of **true** or **false**. If **<p>** contains variables, **ask** obtains variable bindings from the user. See **output**.

asks **(asks <p>)**
calls **output**, prints the result, and reads the user's answers. If **<p>** is a ground proposition, **asks** tries to obtain an answer of **true** or **false**. If **<p>** contains variables, **asks** obtains a list of binding lists from the user. See **output**.

assert **(assert <p>)**
stores the proposition **<p>** in the data base and performs all appropriate forward inference. **Assert** is an abstract operator implemented using **kb** and **toassert**.

assert-and **(assert-and (and $<p_1>$ ... $<p_n>$))**
separately asserts each of the conjuncts $<p_1>, \ldots, <p_n>$.

assert-iff **(assert-iff (iff <p> <q>))**
asserts **(if <p> <q>)** and **(if <q> <p>)**.

assumable **(assumable <p>)**
means that the proposition **<p>** can be assumed if necessary in trying to prove a proposition. See **residue**, **residues**.

bagof **(bagof <x> <p> <s>)**
means that **<s>** is the bag of all objects **<x>** that statisfy **<p>**. Since there maybe many ways of satisfying **<p>**, the bag **<s>** may contain duplicate objects. **Bagof** is useful for performing extensional reasoning, since it allows one to designate the set of all solutions to a problem. See **truep-bagof**, **lookup-bagof**.

batchp **(batchp <x> <y>)**
checks whether the expressions **<x>** and **<y>** can be unified by some set of bindings for the base-level variables in the two expressions. If so, **batchp** returns the corresponding binding list for the variables in **<x>** but discards the bindings for the variables in **<y>**. If the expressions are not unifiable, **batchp** returns **nil**. All variables in **<x>** are treated as distinct from the variables in **<y>**, even though they have the same name. For example, the expression **(r $x b)** matches **(r a $x)** with result **(($x . a) (t . t))**. See **blvarp**, **matchp**.

bc **(bc <p>)**
tries to prove the proposition **<p>**. If successful, it returns an appropriate binding list; otherwise, it returns **nil**. Only base-level variables are treated as variables by **bc**, and any meta-level variables are treated as constants. The inference procedure used is backward chaining, but there are also built-in procedural attachments for many propositions (specified via the **totruep** and **totrueps** relations). **Bc** is implemented using the subroutine **bcdisp**. See **bcdisp**, **scheduler**.

bcdisp **(bcdisp <g1> <a1> <j1> <ce>)**
performs one backward chaining step in trying to prove the propositions on the goal list **<g1>**. The binding list **<a1>** holds bindings for the variables in **<g1>** obtained in preceding steps. The justification list **<j1>** holds the names of any propositions used in deriving a goal list from its super goal list. The list **<ce>** is a stack of supergoals and justifications. In working on a goal list **(<q> . <l>)**, **bcdisp** first calls **trtruep** to find any procedural attachments for **<q>** other than **bc** or **bcs**, and if successful calls that subroutine. Otherwise, it generates subgoals by looking in the data base for propositions of the form **<q>** or **(if <p> <q>)**. The order in which multiple **bcdisp** tasks are executed can be influenced via appropriate **preferred** propositions. **bcdisp** caches its results and saves justifications as appropriate. See **trtruep**, **totruep**, **totrueps**, **cache**, **justify**.

bcs **(bcs <p>)**
tries to prove the proposition **<p>**. It returns a list of all binding lists for which it is successful. Only base-level variables are treated as variables by **bcs**, and any meta-level variables are treated as constants. The inference procedure used is backward chaining, but there are also built-in procedural attachments for many propositions (specified via the **totruep** and **totrueps** relations). **Bcs** is implemented using the subroutine **bcdisp**. See **bcdisp** and **scheduler**.

blvarp **(blvarp <xp>)**
returns a non-nil value if **<xp>** is a base-level variable and otherwise returns nil. A base-variable in MRS is denoted by a dollarsign prefix **($)** and is internally distinguished by the value **bl**. For example, **$a** is a base-level variable. See **varp**.

br **(br <p>)**
tries to prove the proposition **<p>**. If succesful, it returns a list of assumable propositons which, when added to the data base, imply **<p>**. Only base-level variables are treated as variables by **br**, and any meta-level variables are treated as constants. The inference procedure used is backward chaining, but there are also built-in procedural attachments for many propositions (specified via the **totruep** and **totrueps** relations). **Br** is implemented using the subroutine **brdisp**. See **brdisp**, **scheduler**, and **assumable**.

brdisp **(brdisp <g1> <a1> <th> <j1> <ce>)**
performs one backward chaining step in trying to find a residue for the propositions on the goal list **<g1>**. The binding list **<a1>** holds bindings for the variables in **<g1>** obtained in preceding steps. The theory **<th>** contains all assumptions made so far. The justification list **<j1>** holds the names of any propositions used in deriving a goal list from its super goal list. The list **<ce>** is a stack of supergoals and justifications. In working on a goal list **(<q> . <l>)**, **brdisp** first calls **trtruep** to find any procedural attachments for **<q>** other than **bc** or **bcs**, and if successful calls the subroutine so found. Otherwise, it generates subgoals by looking in the data base for propositions of the form **<q>** or **(if <p> <q>)**. It also uses **trtrueps** to discover whether **q** is assumable. If it is assumable and if it is a ground proposition after plugging in the variable bindings returned by **trtrueps**, **brdisp** creates a new theory that includes **<th>**, asserts the proposition in that theory, and generates appropriate subgoals. The asserted propositions are useful in that they make possible consistency checking before making assumptions in subsequent steps. The order in which multiple **brdisp** tasks are executed can be influenced via appropriate **preferred** propositions. **Brdisp** caches its results and saves justifications as appropriate. See **trtruep**, **totruep**, **totrueps**, **cache**, **justify**.

brs **(brs <p>)**
tries to prove the proposition **<p>**. It returns a list of all assumption lists for which it is successful. Only base-level variables are treated as variables by **brs**, and any meta-level variables are treated as constants. The inference procedure used is backward chaining, but there are also built-in procedural attachments for many propositions (specified via the **totruep** and **totrueps** relations). **Brs** is implemented using the subroutine **brdisp**. See **brdisp**, **scheduler**, and **assumable**.

cache
is a variable governing whether various inference methods should cache their results. When non**NIL**, those various inference methods will call the appropriate **tocache** method on each cachable result. (I.e. each method will call **(kb 'tocache <p>)** for each intermediate conclusion, **<p>**.) See **tocache**, **cachebystash**.

cachebystash **(cachebystash <p>)**
stashes the value **<p>** into the theory named by the variable **cache**. (If **cache** has the value **T**, then the current theory is used.) This **cachebystash** subroutine is the default caching method. It is recommended that one house these propositions in a temporary theory, and apply **empty** to this theory when the cached values are no longer needed. See **cache**, **tocache**

characteristic **(characteristic <set> <fn>)**
means that the lisp subroutine **<fn>** is the characteristic function for the set **<set>**, e.g. **(characteristic integers fixp)**. See **arity**, **domain**.

cnf

is short for conjunctive form. A proposition is in conjunctive form if it is written as a conjunction of disjunctions of literals, i.e. atomic propositions or negations of atomic propositions. For example, the proposition
(and (or (not (p $x)) (q $x)) (or (r $x) (s $x))) is in conjunctive form. This also serves as a representation. See **repn**.

cnf-assert **(cnf-assert <p>)**
converts **<p>** into conjunctive normal form and separately asserts each of the conjuncts. See **cnf**.

cnf-unassert **(cnf-unassert <p>)**
converts **<p>** into conjunctive form and separately unasserts each of the conjuncts. See **cnf**.

computable-repn [Concept]
Many relations and functions can be readily evaluated, and so never need to be explicitly stashed. Consider, for example, the class of arithmetic functions and relations, such as + and >. MRS includes several computable representations in which to encode such facts.

We describe below various computable representations -- viz., **re**, **rq**, **rqb**, **rqbm**, **reb**, **rebm**, **rqfm**, **fe**, **fq** and **fea**. For a proposition to be represented in one of these representation, its relation symbol must have an associated LISP subroutine (ALS). Each lookup subroutine associated with each of these representations takes as input a proposition of the form **(<r> <x_1> .. <x_n>)**, and calls ALS on a list of values computed from that argument list, **<x_1> ... <x_n>**. In some representations, each argument is first evaluated, using **lookupval**. Also, in some representations the ALS takes all **n** arguments, while in others it is only passed the term-part of the proposition, namely the first **n-1** arguments. Note that propositions stored in this way are not associated with any particular theory and cannot be found by **PR**-based routines like **prfacts** or **prcontents**. The details of these representations are specified in the **Computable-Repn [Relation]** and **Computable-Repn [Function]** entries. These representations are used by the **funproc** and **relnproc** relations.

computable-repn [Function]

Here we describe those **computable-repn** representations which are based on a function. We designate these function-based representations by using the letter F in the first position of the name of the representation -- e.g. **Fq** is a function-based representation. Here, only the term part of the proposition is passed to the ALS; and it is the responsibility of the associated lookup subroutines to bind this returned value appropriately.

The same second letter is E for Eval, Q for Quote convention used for the relation-based representations applies here as well. Hence lookup subroutines associated with the **FE** representation will first **lookupval** each of the arguments $\langle x_1 \rangle \ \ldots \ \langle x_{n-1} \rangle$, passing the resulting list to the ALS.

The only (current) additional letter for function-based representations is A, for arithmetic. This uses **NUM-=** rather than **UNIFYP** when comparing the value associated with the term of the proposition with the value of the proposition. See **computable-repn**, **fe**, **fea**, **fq**.

computable-repn [Relation]

the **computable-repn** representations are based on relation. These relation-based representations are designated by using the letter R in the first position of the name of the represenation -- e.g. **Ero** is a relation-based representation. With one exception, the ALS takes a spread version of the full proposition as its arguments. If the second letter is E for eval, (as in **rEbm**,) the associated lookup methods will first call lookupval on each embedded term, and pass that evaluated argument list to the ALS. Otherwise, when it is Q for quote, those argument are directly passed to the ALS. By default, the value returned by the ALS is an arbitrary value which, when non**NIL**, tells the lookup subroutine that this propostion is true.

The third and fourth letter encode further refinements: When the third letter in the representation's name is B, (e.g. **reB**,) the ALS itself will return a binding list, which the lookup subroutine will return. For these **r?b** relations, a subsequent M (e.g. **rebM**,) means the ALS returns a list of binding-lists, rather than just one. This 4th letter, M, means multiple values convention is retained for the RQFM representation (used for multi-functions). Here the ALS takes only the term part of the proposition, and returns a set of values. The **RQFM-LOOKUPS** subroutine then forms the list of appropriate binding-lists. [Exception mentioned.] Consider the Square-Root multi-function, which returns both the + and - root of a number. See **Computable-Repn [Concept]**, **repn**, **re**, **reb**, **rebm**, **rq**, **rqb**, **rqbm**, **rqfm**.

contents **(contents <t>)**

returns a list of propositions stored in theory **<t>**. Only those facts stored using the propositional representation (i.e., **pr**) will be found.

cut **(cut)**

is a special control form. When **(cut)** is executed, all other subtasks of the enclosing **doable** or **undoable** task are discarded. As a result, if the subtask containing the **(cut)** form fails, the enclosing **doable** or **undoable** subtask will fail as well. See **doable**, **undoable**.

datum **(datum <x>)**
returns the symbol corresponding to the expression or proposition **<x>**.

deactivate **(deactivate $<t_1>$ ... $<t_n>$)**
deactivates the named theories. See **theory**, **activetheories**, **activate**.

def **(def <k> <k1> . . . <kn>)**
means that the task **<k>** is defined as **(doand <k1> . . . <kn>)**. A task can have more than one definition, each one covering a different set of inputs. For example, the following propositions define the factorial function.

```
(def (fact 0 1))]
(def (fact &m &n)
     (- &m 1 &p)
     (fact &p &q)
     (* &m &q &n))
```

defobject **(defobject <name> $<p_1>$ ... $<p_n>$)**
unasserts all propositions in **pr** that mention **<name>** and then asserts the propositions $<p_1>$. . . .. $<p_n>$.

deftheory **(deftheory <name> $<p_1>$ ... $<p_n>$**
empties the theory **<name>** and asserts propositions $<p_1>$. . . .. $<p_n>$ into it.

disjoint **(disjoint <x> <y>)**
means that lists **<x>** and **<y>** do not have any elements in common.

```
(disjoint nil $y)
(disjoint $x nil)
(if (and (not (element $e $s)) (disjoint $l $s))
    (disjoint ($e . $l) $s))
```

Procedural attachment: **truep-disjoint**. The lisp file **set** must be loaded from the mrs directory.

disqualified **(disqualified <k>)**
states that the task **<k>** is disqualified. A task that is **applicable** is **executable** unless it is **disqualified**. The chief way a task can get disqualified is for there to be another **applicable** task that is **preferred** to it. However, this fact is used when either of the switches **executable** or **preferred** is non-nil. See **applicable**, **scheduler**.

dl **(repn <p> dl)**
means that the proposition **<p>** should be represented in the **dl** representation, i.e. the **dl-<x>** family of subroutines will be used to stash, unstash, and lookup **<p>**. This representation is particularly useful for representing propositions involving non-functional binary relations, e.g. **(neighbor france switzerland)**. Note that propositions stored in this way are not associated with any particular theory and cannot be found by PR-based routines like **prfacts** or **prcontents**. See **dl-lookup**, **dl-stash**, **dl-unstash**, **repn**.

dl-lookup **(dl-lookup (<r> <a> <b>))**
matches **<b>** against each of the values stored as the **<r>** property of the lisp atom **<a>**, and if successful returns the resulting binding list. Both **<r>** and **<a>** must be atoms. See **dl**.

dl-lookups **(dl-lookups (<r> <a> <b>))**
matches **<b>** against each of the values stored as the **<r>** property of the lisp atom **<a>**, and returns a list of the binding lists for every successful match. Both **<r>** and **<a>** must be atoms. See **dl**.

dl-stash **(dl-stash (<r> <a> <b>))**
adds **<b>** to the list of values stored as the **<r>** property of the atom **<a>**. Both **<r>** and **<a>** must be atoms. See **dl**.

dl-unstash **(dl-unstash (<r> <a> <b>))**
removes **<b>** from the list of values stored as the **<r>** property of **<a>**. Both **<r>** and **<a>** must be atoms. See **dl**.

dnf
is short for disjunctive form. A proposition is in disjunctive form if it is written as a disjunction of conjunctions of literals, i.e. atomic propositions or negations of atomic propositions. For example, the proposition **(or (and a b) (and c d))** is in disjunctive form. This also serves as a representation. See **repn**.

doable **(doable <k>)**
designates the task of trying to execute the task **<k>**. The task **(doable <k>)** succeeds if there is a successful execution of **<k>**. However, after a single success, all other subtasks of **<k>** are discarded, and so it can succeed at most once. Note that as a result of the current implementation, it is not possible to interleave subtasks outside a **doable** task with those inside. See **succeed** and **cut**.

doall **(doall <x> <k> <s>)**
designates the task of getting all **<x>** for which the task **<k>** succeeds. It packages these into a list and succeeds if the resulting list unifies with **<s>**.

doand **(doand $<k_1>$ . . . $<k_n>$)**
designates the task of executing tasks $<k_1>$, . . ., $<k_n>$ in sequence. If one of the tasks can be executed in more than one way, separate subtasks are set up for each possibility. The **doand** task succeeds if there is at least one succesful execution of all the tasks in the list. For example, **(doand (memlist &x ((a) b (c))) (atom &x t))** succeeds because the **atom** task succeeds for the execution of the **mem** task in which **&x** is bound to **b**. The order in which subtasks are executed can be influenced by making appropriate **preferred** statements.

domain **(domain <rel> <i> <set>)**
provide typing information, indicating that the **<i>**th argument of the relation (or operation) **<rel>** must belong to the set **<set>**. E.g. **(domain stash 1 propositions)**. See **arity, charactistic, theories, terms, propositions**.

door **(door <k_1> . . . <k_n>)**
designates the task of trying to execute one the tasks <k_1>, . . ., <k_n>.

edunit **(edunit <x>)**
allows the user to edit the propositions about <x> using Zwei. [AVAILABLE ONLY IN ZETALISP.]

element **(element <x> <s>)**
means that object <x> is an element of list <s>.

```
(element $e ($e . $1))
(if (element $e $1) (element $e ($y . $1)))
```

Procedural attachment: **truep-element**. The lisp file **set** must be loaded from the mrs directory.

elementsin **(elementsin <b> <s>)**
means that <s> is the set of all elements in the bag <b>.

```
(elementsin nil nil)
(if (and (not (element $e $1)) (elementsin $1 $s))
    (elementsin ($e . $1) ($e . $s)))
(if (and (element $e $1) (elementsin $1 $s))
    (elementsin ($e . $1) $s))
```

Procedural attachment: **truep-elementsin**. The lisp file **set** must be loaded from the mrs directory.

empty **(empty <t>)**
unasserts all the facts in the theory <t>. Only those facts stored using the propositional representation (i.e., **pr**) will be found.

exdisp **(exdisp <l> <al>)**
performs one step in the execution of the list of tasks <l>. The alist <al> is a list of variable bindings obtained so far.

executable **(executable <k>)**
states that the task <k> is executable. If the value of the variable **executable** is non-nil, **scheduler** uses **trtruep** to find a task <k> such that **(executable <k>)** is true. If the value of **executable** is nil, it simply selects an element from the value of the variable **agenda**. If the value of the variable **preferred** is nil, it takes the first element; otherwise, it uses **trtruep** to find the best according to the **preferred** relation. See **scheduler**.

executable
See definition of the meta-level **executable**

execute **(execute <k>)**
tries to execute the task <k> and, if successful, returns a binding list for the meta-level variables in <k>. It is implemented using **exdisp**. See **task**, **def**, **doable**, **undoable**, **doall**, **doand**, **door**, **succeed**, **cut**, **preferred**, **tracetask**.

executed **(executed <k>)**
means that the task **<k>** has been executed. If the value of the variable **executed** is non-nil, **scheduler** will use **trassert** to record this fact of each task as it is performed. See **scheduler**.

executed
See definition of the meta-level **executed**.

executes **(executes <p>)**
tries to execute the task **<k>** and, if successful, returns a list of all binding lists for the meta-level variables in **<k>**. It is implemented using **exdisp**. See **task**, **def**, **doable**, **undoable**, **doall**, **doand**, **door**, **succeed**, **cut**, **preferred**, **tracetask**.

fc **(fc <p>)**
means that proposition **<p>** is assert and then all rules with **<p>** in the premise are checked to see if the premise is true and if so the consequence is asserted. Only base-level variables are treated as variables by **fc**, and any meta-level variables are treated as constants. The inference procedure used is forward chaining, but there are also built-in procedural attachments for many propositions (specified via the **toassert** relation). **Fc** is implemented using the subroutine **fcdisp**. See **fcdisp**, **scheduler**.

fcdisp **(fcdisp <p>)**
performs one forward chaining step on the proposition **<p>**. **Fcdisp** first calls **trtruep** to find a procedural attachments for **<p>** other than **fc**, and if successful calls that subroutine. Otherwise, it generates other assertions by looking in the data base for propositions of the form **(if <p> <q>)** and **(if (and ... <p> ...) <q>)**. The order in which multiple **fcdisp** tasks are executed can be influenced via appropriate **preferred** propositions. **Fcdisp** caches all results and saves justifications as appropriate. See **trtruep**, **toassert**, **justify**.

fe **(repn <p> fe)**
means that the proposition **<p>** should be represented in the **fe** representation. That is, the **fe-<x>** family of subroutines will be used to retrieve **<p>**. Its lookup method, **fe-lookup**, takes as its argument a proposition of the form **(<f> <x$_1$> ... <x$_n$> <x$_{n+1}$>)**. It first evaluates the term **(<f> <x$_1$> ... <x$_n$>)** by applying LISP subroutine corresponding to the function symbol **<f>**, (i.e., on **<f>**'s **LISP** property,) to the list obtained by calling **lookupval** on each of the **<x$_1$>**. **Fe-Lookup** then unifies this value with **(lookupval <x$_{n+1}$>)**, and returns the result. See **computable-repn**, **fe-lookup**, **fe-lookups**, **fea**, **fq**, **funproc**, **repn**.

fe-lookup **(fe-lookup <p>)**
is used to retrieve the proposition **<p>**. See **fe**, **lisp**.

fe-lookups **(fe-lookups <p>)**
is functionally equivalent to **(pluralize (Fe-lookup p))**. See **fe**, **fe-lookup**.

fea **(repn <p> fea)**
means that the proposition <p> should be represented in the **fea** representation. That is, the **fea-<x>** family of subroutines will be used to retrieve <p>. Its lookup method, **fea-lookup**, takes as its argument a proposition of the form (<f> $<x_1>$ ... $<x_n>$ $<x_{n+1}>$). It first evaluates the term (<f> $<x_1>$ ... $<x_n>$) by applying LISP subroutine corresponding to the function symbol <f>, (i.e., on <f>'s **LISP** property,) to the list obtained by calling **lookupval** on each of the $<x_1>$. **Fea-Lookup** then compares this value to (**lookupval** $<x_{n+1}>$) using **num-=**, and returns the result. (**Fe-lookup** just uses **unifyp** to produce this comparison.) See **computable-repn**, **fea-lookup**, **fea-lookups**, **fe**, **funproc**, **repn**.

fea-lookup **(fea-lookup <p>)**
is used to try to retrieve the proposition <p>. See **fea**, **num-=**, **lisp**.

fea-lookups **(fea-lookups <p>)**
is functionally equivalent to (**pluralize** (**Fea-lookup p**)). See **fea**, **fea-lookup**.

fq **(repn <p> fq)**
means that the proposition <p> should be represented in the **fq** representation. That is, the **fq-<x>** family of subroutines will be used to retrieve <p>. By default, most functions, (including arithmetic ones,) use this representation. Its lookup method, **fq-lookup**, takes as its argument a proposition of the form (<f> $<x_1>$ ... $<x_n>$ $<x_{n+1}>$). It first evaluates the term (<f> $<x_1>$ ... $<x_n>$) by applying LISP subroutine corresponding to the function symbol <f>, (i.e., on <f>'s **LISP** property,) to the list $<x_1>$ ... $<x_n>$. **Fq-Lookup** then unifies this value with $<x_{n+1}>$, and returns the result. See **computable-repn**, **fe**, **fq-lookup**, **fq-lookups**, **fqm**, **repn**.

fq-lookup **(fq-lookup <p>)**
is used to try to retrieve the proposition <p>. See **fq**, **lisp**.

fq-lookups **(fq-lookups <p>)**
is functionally equivalent to (**pluralize** (**Fq-lookup p**)). See **fq**, **fq-lookup**.

function **(function <x>)**
means that <x> is a function. See **singlep**

funproc [2] **(funproc <sym> <op>)**
means that the operator <op> is procedurally attached to the symbol <sym>. E.g. after asserting (**funproc + plus**), (**lookupval (+ 2 3 4)**) will pass **2**, **3** and **4** to the LISP procedure **plus**, and return its answer, **9**. Note that this looking-up mechanism is (intentionally) very simple -- while (**lookup '(+ 2 3 4 $a)**) will work (returning a binding list which includes (**$a . 9**),) both (**lookup '(+ 2 3 $b 9)**) and (**lookup '(+ $d 0 $c)**) will return **nil**. The (**funproc <sym> <op>**) assertion will use **funproc-assert** to forward chain to assert (**function <sym>**), and that all propositions (and terms) whose relation symbol is <sym> should use the **fq** representation. See **fq**, **function**, **funproc [3]**, **funproc-assert**, **relnproc**.

funproc [3] **(funproc <sym> <op> <repn>)**
This is an embellishment of the binary **funproc**, listed under **funproc [2]**. After a **(funproc + plus)** assertion, both **(lookup '(+ 2 (+ 3 4) $x))** and **(lookup '(+ 2 3 4) 9.0))** will fail -- i.e. return **nil**. One can use the (optional) third argument, **<repn>**, to permit two other, more elaborate forms of functional procedural attachment. The **(funproc + plus FE)** assertion handles the first problem. Now each embedded ground term -- here **2** and **(+ 2 3)** -- will first be **lookupval**ed, and the result passed to the LISP procedure **plus**. That is, this uses the **fe** representation, rather than **fq**. The assertion **(funproc + plus FEA)** solves the second problem, causing **(+ . &x)** to use the **fea** representation.

The **<repn>** term defaults to **fq** if omitted. One can also substitute **EVAL** for **fe**, or **=** for **fea**. See **fea**, **fe**, **fq**, **function**, **funproc [2]**, **funproc-assert**, **num-=**, **relnproc**.

funproc-assert **(funproc-assert (funproc <sym> <op> <repn>))**
asserts the proposition **(funproc <sym> <op> <repn>)**. (The same subroutine is used to assert **(funproc <sym> <op>)**.) See **funproc**.

getbdg **(getbdg <v> <p>)**
is equivalent to **(getvar <v> (truep <p>))**.

getbdgs **(getbdgs <v> <p>)**
is equivalent to **(mapcar '(lambda (1) (getvar <v> 1)) (trueps p))**.

getval **(getval (<r> $<x_1>$ ... $<x_n>$))**
is equivalent to **(getbdg <y> (<r>** $<x_1>$ ... $<x_n>$ **<y>))**.

getvals **(getvals (<r> $<x_1>$ ... $<x_n>$))**
is equivalent to **(getbdgs <y> (<r>** $<x_1>$ ... $<x_n>$ **<y>))**.

getvar **(getvar <v> <l>)**
looks up the binding of the variable **<v>** on the binding list **<l>**, fully instantiates it with respect to the other variables on **<l>**, and returns the result. For example, **(getvar '$x '(($x . (f $y)) ($y . a)))** would return **(f a)**.

ground **(ground <x>)**
states that the expression **<x>** is a ground expression, i.e. it contains no variables. See **lookup-ground**.

groundp **(groundp <x>)**
returns **t** if and only if the expression **<x>** contains no variables.

if **(if <p> <q>)**
means that whenever proposition **<p>** is true, proposition **<q>** is true. See **bc**, **br**, **fc**.

iff **(iff <p> <q>)**
is equivalent to **(and (if <p> <q>) (if <q> <p>))**.

includes `(includes <c> <d>)`
means that the theory `<c>` includes the theory `<d>`.

includes `(includes <t1> <t2>)`
makes theory $\langle t_1 \rangle$ a supertheory of theory $\langle t_2 \rangle$ so that whenever $\langle t_1 \rangle$ is active $\langle t_2 \rangle$ will be active as well. In effect theory $\langle t_1 \rangle$ includes all of the propositions in $\langle t_2 \rangle$. Both **theory** and **activetheories** take these inclusions into account.

indb `(indb <p>)`

indbp `(indbp <p>)`
means that a proposition equal to `<p>` up to variable renaming is stored in the **pr** representation. See **pr-indbp**.

integer `(integer <x>)`
means that `<x>` is an integer.

inter `(inter <x> <y> <b>)`
means that list `<b>` is every element in list `<x>` that is in list `<y>`.

```
(inter nil $y nil)
(if (and (element $e $y) (inter $1 $y $s))
    (inter ($e . $1) $y ($e . $s)))
(if (and (not (element $e $y)) (inter $1 $y $s))
    (inter ($e . $1) $y $s))
```

Procedural attachment: **truep-inter**. The lisp file **set** must be loaded from the mrs directory.

intersect `(intersect <x> <y>)`
means that bag `<x>` and bag `<y>` have an equal element.

```
(if (or (element $e $s) (intersect $1 $s))
    (intersect ($e . $1) $s))
```

Procedural attachment: **truep-intersect**. The lisp file **set** must be loaded from the mrs directory.

is `(is <x> <y>)`
means that the value of the arbitrarily nested expression `<x>` is `<y>`. See **lookup-is**, **truep-is**.

just `(just <q> <m> <p1> ... <pn>)`
means that the justification for the proposition named `<q>` is the inference method `<m>` and the premises $\langle p_1 \rangle, \ldots \langle p_n \rangle$. See **where**, **why**, **justify**, **tm-unassert**.

justify

is a variable governing MRS's mechanism for recording justifications. When non**NIL**, its value is the name of the theory into which MRS will save justifications for all deductions. (If **justify** has the value **T**, then the current theory is used.) It is recommended that one house these propositions in a temporary theory, and apply **empty** to this theory when these justifications are no longer needed. See **why** and **where**.

kb **(kb to<g> <arg$_1$> ... <arg$_N$>)**
is MRS's way of handling procedural attachments. It is equivalent to **(apply (getvar '&f (trtruep (to<g> <arg$_1$> ... <arg$_N$> &f) <arg$_1$>...<arg$_N$>)))**. See **to<g>**.

length **(length <l> <n>)**
means that the list **<l>** is of length **<n>**.

lhfalse **(lhfalse (unprovable <p>))**
calls **lookup** on the proposition **<p>**. It returns **nil** if the answer is non-nil; otherwise, it returns **truth**.

lhtrue **(lhtrue (provable <p>))**
calls **lookup** on the proposition **<p>** and returns the answer.

lisp **(lisp <sym> <op>)**
means that **<op>** is the Lisp subroutine used to compute the function denoted by the symbol **<sym>**. E.g. **(lisp + plus)**. See **Computable-Repn [Concept]**, **fea-lookup**, **fe-lookup**, **fq-lookup**, **re-lookup**, **rq-lookup**, **rqb-lookup**, **reb-lookup**, **rqfm-lookups**.

lookup **(lookup <p>)**
checks whether the proposition **<p>** matches a proposition in the data base and, if so, returns the correponding binding list. **Lookup** is an abstract operator implemented using **kb** and **tolookup**.

lookup-= **(lookup-= (= <x> <y>))**
calls **unifyp** on the expressions **<x>** and **<y>** and returns the result. See **=**.

lookup-bagof **(lookup-bagof (bagof <x> <p> <s>))**
calls **lookups** on **<p>** and matches **<s>** against the sequence formed by plugging the answers into **<x>**. **Lookup-bagof** is useful for performing extensional reasoning, since it allows one to designate the set of all solutions to a problem.

lookup-ground **(lookup-ground (ground <x>))**
returns **((t . t))** if and only if the expression **<x>** contains no variables. See **ground**.

lookup-is **(lookup-is (is <x> <y>))**
uses **lookupval** to evaluate the arbitrarily nested expression **<x>** and tries to unify the answer with **<y>**. See **is**.

lookupapplicable **(lookupapplicable (applicable <k>))**
tries to match **<k>** with each element of **agenda** and returns a corresponding binding list if successful. See **applicable**.

lookupbdg **(lookupbdg <v> <p>)**
is equivalent to **(getvar <v> (lookup <p>))**.

lookupbdgs **(lookupbdgs <v> <p>)**
is equivalent to **(mapcar '(lambda (x) (getvar <v> x)) (lookups <p>))**.

lookupbylookups **(lookupbylookups <p>)**
is equivalent to **(singularize (lookups <p>))**.

lookups **(lookups <p>)**
checks whether the proposition **<p>** matches any propositions in the data base and returns a list of binding lists for each successful match. **Lookups** is an abstract operator implemented using **kb** and **tolookups**.

lookupsapplicable **(lookupsapplicable (applicable <k>))**
tries to match **<k>** with each element of **agenda** and returns list of binding lists for each successful match. See **applicable**.

lookupsbylookup **(lookupsbylookup <p>)**
is equivalent to **(pluralize (lookup <p>))**.

lookupval **(lookupval (<f> $<x_1>$ ... $<x_n>$))**
is equivalent to **(getvar <y> (lookup (<f> $<x_1>$ ... $<x_n>$ <y>)))**.

lookupvals **(lookupvals (<f> $<x_1>$ ... $<x_n>$))**
is equivalent to
(mapcar '(lambda (x) (getvar <y> x)) (lookups (<f> $<x_1>$ ... $<x_n>$ <y>))).

mand **(mand <p> <l>)**
is satisfied if **(<p> x)** is true for every **x** in the list **<l>**.

```
(mand $p nil)
(if (and ($p $x) (mand $p $l))
    (mand $p ($x . $l)))
```

Procedural attachment: **truep-mand**. The lisp file **set** must be loaded from the mrs directory.

mandcan **(mandcan <p> <l> <s>)**
means that **<s>** is the union of the lists **y** that satisfy **(<p> x y)** for every element **x** in list **<l>**.

```
(mandcan $f nil nil)
(if (and ($f $x $y) (mandcan $f $l $s) (union $y $s $t))
    (mandcan $f ($x . $l) $t))
```

Procedural attachment: **truep-mandcan**. The lisp file **set** must be loaded from the mrs directory.

mandcar **(mandcar <p> <l> <s>)**
means that **<s>** is the set of objects **y** that satisfy **(<p> x y)** for every element **x** in list **<l>**.

```
(mandcar $f nil nil)
(if (and ($f $x $y) (mandcar $f $l $s))
    (mandcar $f ($x . $l) ($y . $s)))
```

Procedural attachment: **truep-mapcar**. The lisp file **set** must be loaded from the mrs directory.

matchp **(matchp <x> <y>)**
checks whether the expressions **<x>** and **<y>** can be unified by some set of bindings for the meta-level variables in the two expressions. If so, **matchp** returns the corresponding binding list for the variables in **<x>** but discards the bindings for the variables in **<y>**. If the expressions are not unifiable, **matchp** returns **nil**. All variables in **<x>** are treated as distinct from the variables in **<y>**, even though they have the same name. For example, the expression **(r &x b)** matches **(r a &x)** with result **((&x . a) (t . t))**. See **batchp**.

mem **(mem <e> <c>)**
means that the element <e> is a member of the set of <c>. E.g., **(mem george people)**. See **subclass**.

member **(member <a> <s>)**
means that **<a>** is a member of the list **<s>**, e.g. **(member 5 (4 5 6))**.

memlist **(memlist <x> <l>)**
designates the task of checking whether the object **<x>** is in the list **<l>**. **Memlist** tries to unify **<x>** with each element of **<l>** and succeeds once for each match that it finds.

mlvarp **(mlvarp <xp>)**
returns a non-nil value if **<xp>** is a meta-level variable and otherwise returns nil. A meta-variable in MRS is denoted by an ampersand prefix **(&)** and is internally distinguished by the value **ml**, e.g. **&a** is a meta-level variable. See **varp**.

mrsapropos **(mrsapropos <s>)**
returns a list of all LISP atoms containing **<s>** as a substring of their printnames.

mrsdemo **(mrsdemo)**
presents a demonstration of the range of capabilities in MRS.

mrsdescribe **(mrsdescribe <x>)**
prints out the portion of this dictionary relevant to the object **<x>**.

mrsdump **(mrsdump <t> <f>)**
saves the propositions from theory **<t>** in a form that allows them to be reloaded with LISP's **load** command. Only those facts stored using the propositional representation (i.e., **pr**) will be found.

mrshelp **(mrshelp <k>)**
provides information about the MRS keyword **<k>**.

mrsload **(mrsload <f>)**
loads a file **<f>** of propositions.

mrssave **(mrssave <t_1> ... <t_n> <f>)**
saves the propositions from theories **<t_1>** ... **<t_n>** in the file **<f>** in a form that allows them to be reloaded with the **mrsload** command. Note that this works only for propositions stored in the **pr** representation.

mrstofunctions
refers to the set of all MRS **to<g>** functions -- e.g. **(mem tostash mrstofunctions)**. See **domain**.

not **(not <p>)**
means that the proposition **<p>** is false. This is not equivalent to **unknown**, or to **unprovable**.

num-= **(num-= <x> <y>)**
means that the expressions **<x>** and **<y>** are numerically equal -- or close enough to qualify. Procedurally, if **<x>** and **<y>** (are nonNIL and) unify, that MGU value is returned. Otherwise, if both terms are ground atomic numeric expressions, whose difference is less than **NUM-=-THRESHOLD**, **truth** is returned. E.g. **(num-= 3 3.0)** returns **truth**, whereas **(unifyp 3 3.0)** returns **nil**. See **fea**, **num-=-theshold**.

num-=-threshold
is a special variable whose value is the tolerance required for two numeric values to be considered equal. It is initially set to **0.0001**. See **num-=**.

number **(number <x>)**
means that **<x>** is a number.

or **(or <p_1> . . . <p_n>)**
means that one or more of the propositions **<p_1>** . . . **<p_n>** is true.

output **(output <x>)**
translates the expression **<x>** into pseudo-natural language in accordance with programmer-defined templates. See **template**.

pattern **(pattern <d>)**
returns the proposition corresponding to the proposition symbol **<d>**.

perceive **(perceive <p>)**
determines whether the proposition **<p>** is true by direct observation rather than inference. **Perceive** is an abstract operator implemented using **kb** and **toperceive**.

perceive-indb **(perceive-indb (indb <p>))**

perceive-not **(perceive-not (not <p>))**

perceives **(perceives <p>)**
determines whether the proposition **<p>** is true by direct observation rather than inference and returns a list of all binding lists for which it succeeds. **Perceives** is an abstract operator implemented using **kb** and **toperceives**.

pl **(repn <p> pl)**
means that the proposition **<p>** should be represented in the **pl** representation, i.e. the **pl-<x>** family of subroutines will be used to stash, unstash, and lookup **<p>**. This representation is particularly useful for representing propositions involving unary functions, e.g. **(arity member 2)**. Note that propositions stored in this way are not associated with any particular theory and cannot be found by PR-based routines like **prfacts** or **prcontents**. See **pl-lookup**, **pl-stash**, **pl-unstash**, **repn**.

pl-lookup **(pl-lookup (<f> <a> <b>))**
matches **<b>** against the **<f>** property of the lisp atom **<a>**. **<f>** and **<a>** must both be atoms. See **pl**.

pl-stash **(pl-stash (<f> <a> <b>))**
places **<b>** on the property list of **<a>** under the indicator **<f>**. **<f>** and **<a>** must both be atoms. See **pl**.

pl-unstash **(pl-unstash (<f> <a> <b>))**
removes the **<f>** property from the lisp atom **<a>**, if its value was **<b>**. **<f>** and **<a>** must both be atoms. See **pl**.

plug **(plug <x> <l>)**
returns a copy of the expression **<x>** fully instantiated with respect to the variables on the binding list **<l>**. For example, **(plug '(r $x $z) '(($x . (f $y)) ($y .a)))** would return **(r (f a) $z)**.

pluralize **(pluralize <x>)**
returns the plural-value of **<x>**. That is, it returns **(list <x>)** if **<x>** is non**NIL**, or **nil** otherwise. See **lookupsbylookup**, **truepsbytruep**.

pr **(repn <p> pr)**
means that the proposition **<p>** should be represented in the **pr** representation, i.e. the **pr-<x>** family of subroutines will be used to stash, unstash, and lookup **<p>**. This is MRS's default representation. Propositions stored in this way can be associated with any number of theories and will be available for use only when one or more of those theories are active. See **pr-lookup**, **pr-indbp**, **pr-stash**, **pr-unstash**.

pr-indbp **(pr-indbp <p>)**
checks whether there is a proposition in the **pr** representation that is identical to **<p>** up to variable renaming and, if so, returns its proposition symbol. See **pr**.

pr-lookup **(pr-lookup <p>)**
uses **indexp** and **batchp** to find any matching proposition in the **pr** representation and, if successful, returns the corresponding binding list. See **pr**.

pr-lookups **(pr-lookups <p>)**
uses **indexp** and **batchp** to find any matching proposition in the **pr** representation and returns a list of all binding lists for which it is successful. See **pr**.

pr-stash **(pr-stash <p>)**
stores <p> in the propositional data base and returns the corresponding proposition symbol. See **pr**.

pr-unstash **(pr-unstash <p>)**
removes the proposition <p> from the propositional data base. See **pr**.

prcontents **(prcontents <th>)**
prints out all propositions in theory <th>. Only those facts stored using the **pr** representation will be found.

preferred **(preferred <j> <k>)**
states that the task <j> is preferred to the task <k>. The **preferred** is important in that <k> is disqualified whenever there is an **applicable** task that is **preferred** to it. This relation is the primary way of influencing task ordering in MRS. It has effect only when one of the switches **executable** or **preferred** has a non null value. See **disqualified**, **scheduler**.

prfacts **(prfacts <n>)**
prints out all propositions about <n> in the currently active theories. Only those facts stored using the **pr** representation will be found.

primitive **(primitive <k>)**
states that the operator in the task <k> is a primitive machine operation, i.e. a Lisp subroutine.

property **(property <x> <y> <z>)**
means that the atom <x> has <y> as its <z> property.

propositions **(domain <x> <i> propositions)**
means that the <i>th argument to the subroutine <x> should be a proposition. See **domain**.

re **(repn <p> re)**
means that the proposition <p> should be represented in the **re** representation. That is, the **re-**<x> family of subroutines will be used to retrieve <p>. Its lookup method, **re-lookup**, takes as its argument a proposition of the form (<r> $<x_1>$ ... $<x_n>$) and calls **lookupval** on each of the $<x_i>$. If there is a LISP subroutine (i.e. on <r>'s **LISP** property,) corresponding to the relation symbol <r>, **re-lookup** applies the subroutine and returns **((t . t))** if that subroutines returns non**NIL**: otherwise it return **nil**. See **computable-repn**, **repn**, **re-lookup**, **re-lookups**, **rq**, **reb**, **relnproc**.

re-lookup **(re-lookup <p>)**
is used in trying to retrieve the proposition **<p>**. See **re**.

re-lookups **(re-lookups <p>)**
is functionally equivalent to **(pluralize (Re-lookup p))**. See **re**, **re-lookup**.

reb **(repn <p> reb)**
means that the proposition **<p>** should be represented in the **reb** representation. That is, the **reb-<x>** family of subroutines will be used to retrieve **<p>**. Its lookup method, **reb-lookup**, takes as its argument a proposition of the form **(<r> <x$_1$> ... <x$_n$>)**. If there is a LISP subroutine (i.e. on **<r>**'s **LISP** property.) corresponding to the relation symbol **<r>**, **reb-lookup** applies the subroutine to these arguments, and returns the result (assumed to be a binding list). E.g. **(repn (unifyp $a $b) reb)**. See **computable-repn**, **repn**, **reb-lookup**, **reb-lookups**.

reb-lookup **(reb-lookup <p>)**
is used in trying to retrieve the proposition **<p>**. See **reb**, **relnproc**.

reb-lookups **(reb-lookups <p>)**
is functionally equivalent to **(pluralize (reb-lookup p))**. See **reb**, **reb-lookup**.

rebm **(repn <p> rebm)**
means that the proposition **<p>** should be represented in the **rebm** representation. That is, the **rebm-<x>** family of subroutines will be used to retrieve **<p>**. Its lookup method, **reb-lookup**, takes as its argument a proposition of the form **(<r> <x$_1$> ... <x$_n$>)**. If there is a LISP subroutine (i.e. on **<r>**'s **LISP** property.) corresponding to the relation symbol **<r>**, **reb-lookup** applies the subroutine to these arguments, and returns the result (assumed to be a list of binding lists). See **computable-repn**, **reb**, **rebm-lookup**, **reb-lookups**, **relnproc**, **repn**.

rebm-lookup **(rebm-lookup <p>)**
is functionally equivalent to **(singularize (reb-lookup p))**. See **rebm**, **reb-lookup**.

relnproc [2] **(relnproc <sym> <op>)**
is a way of procedurally attaching the operation **<op>** to the relation symbol **<sym>**. E.g. after asserting **(relnproc < greater-than)**, **(lookup '(< 2 4))** will pass **2** and **4** to the LISP procedure **greater-than**, and seeing its answer is **nonNIL**, the **lookup** call will return **((t . t))**. (It would otherwise return **NIL**.) The **(relnproc <sym> <op>)** assertion will use **relnproc-assert** to forward chain to assert that all propositions whose relation symbol is **<sym>** should use the **rq** representation. See **funproc**, **relnproc [3]**, **relnproc-assert**, **rq**.

relnproc [3] **(relnproc <sym> <op> <repn>)**
There are various possible limitations with binary **relnproc** mechanism, listed above under **relnproc [2]**. First, after asserting **(relnproc < greater-than)**, the query **(lookup '(< 2 (+ 1 3)))** will return **nil**. Second, after asserting **(relnproc unify unifyp)**, **(lookup '(unify (a $y) ($x b)))** will only return **((t . t))**, ignoring the **($x . a)** and **($y . b)** bindings. The (optional) third argument above, **<repn>**, permits other, more elaborate forms of relational procedural attachment. The **(relnproc < greater-than RE)** assertion handles the first problem. Here each embedded ground term -- here **2** and **(+ 1 3)** -- will first be **lookupval**ed, and the result passed to the LISP procedure **greater-than**. This uses the **re** representation, rather than **rq**. The **(relnproc unify unifyp RQB)** assertion handles the second problem, as the **(unify &x &y)** facts will now use the **RQB** representation. Similarly **<repn>** can be set to **RQBM**, **REB**, **REQB** or **RQFM**.

If omitted, **<repn>** here defaults to **rq**. One can also use the aliases **EVAL** for **RE**, **BindList** for **RQB**, **MBindList** for **REBM**, **EBindList** for **REB**, **MEBindList** for **REBM**, or **MultiFn** for **RQFM**. See **relnproc [2]**, **relnproc-assert**, **re**, **reb**, **rebm**, **rq**, **rqb**, **rqbm**, **rqfm**.

relnproc-assert **(relnproc-assert (relnproc <sym> <op> <repn>))**
asserts the proposition **(relnproc <sym> <op> <repn>)**. (The same subroutine is used to assert **(relnproc <sym> <op>)**.) See **relnproc**.

repn **(repn <p> <rpn>)**
means that the representation **<rpn>** should be used to store and access the proposition **<p>**. (See **repns** entry for list of allowable representations.) See **achieve**, **repn-assert**, **repn-method**, **repn-unassert**, **repns**.

repn-assert **(repn-assert (repn <prop> <rpn>))**
uses the **repn-method** declarations associated with this representation **<rpn>** to stash (in a forward chaining manner) the appropriate **to<x>** statements for this **<prop>**. The **domain** specification of that operation is used to determine the exact form of the assertion. E.g. calling **repn-assert** on the assertion **(repn (father &c &d) p1)** will generate the statements **(tolookup (father &c &d) p1-lookup)**, and **(tolookups (father &c &d) p1-lookups)**, as **(domain lookup 1 propositions)**. (Later it may deal with terms, and assert facts like **(tolookupval (father &c) p1-lookupval)**, (via **(domain lookupval 1 terms)**) as well.) See **domain**, **repn**, **repn-method**, **repn-unassert**.

repn-method **(repn-method <rpn> <op> <mthd>)**
means that the LISP subroutine **<mthd>** should be used to perform the **<op>** operation, in the representation **<rpn>**. E.g. **(repn-method pr tostash pr-stash)**. See **repn-assert**, **repn**, **repn-unassert**.

repn-unassert **(repn-unassert (repn <prop> <rpn>))**
undoes the effects (read 'stash'es) of **repn-assert**. See **domain**, **repn**, **repn-assert**, **repn-method**.

repns **(mem <r> repns)**
means that the symbol **r** refers to a representations. Currently existing representations include **cnf**, **dl**, **pl**, **pr**, **tl** and **achieve-perceive**, all of which store results; and **fe**, **fea**, **fq**, **re**, **reb**, **rebm**, **rq**, **rqb**, **rqbm** and **rqfm** which do not. See **repn**, **computable-repn**.

residue **(residue <p>)**
tries to prove the proposition **<p>**. It differs from **truep** in that it is allowed to make assume any proposition asserted to be **assumable**; and, if it is successful in proving **<p>**, it returns a list of its assumptions. The set of assumptions is called the **residue** of **<p>**. **Residue** is an abstract operator defined using **kb** and **toresidue**. See **assumable**.

residues **(residues <p>)**
tries to prove the proposition **<p>**. It differs from **trueps** in that it is allowed to make assume any proposition asserted to be **assumable**; and, if it is successful in proving **<p>**, it returns a list of lists of assumptions. The set of assumptions is called the **residue** of **<p>**. **Residue** is an abstract operator defined using **kb** and **toresidue**. See **assumable**.

resolution **(resolution <p>)**
tries to prove the proposition **<p>**. If successful, it returns an appropriate binding list; otherwise, it returns **nil**. **Resolution** is an implementation of linear-input resolution using the set of support control strategy. In operation, **resolution** negates **<p>** and converts it to conjunctive form, stashes the results in a locally bound theory, and invokes the subroutine **rsdisp** on each conjunct. See **rsdisp**, **scheduler**, **tracetask**, **cnf**.

resolutionresidue **(resolutionresidue <p>)**
tries to prove the proposition **<p>**. If succesful, it returns a list of assumable propositons which, when added to the data base, imply **<p>**. **Resolutionresidue** is an implementation of linear-input resolution using the set of support control strategy. In operation, **resolutionresidue** negates **<p>** and converts to conjunctive form, stashes the results in a locally bound theory, and invokes the subroutine **rrdisp** on each conjunct. See **rrdisp**, **scheduler**, **tracetask**, **cnf**.

resolutionresidue **(resolutionresidue <p>)**
tries to prove the proposition **<p>**. If succesful, it returns a list of assumable propositons which, when added to the data base, imply **<p>**. **Resolutionresidue** is an implementation of linear-input resolution using the set of support control strategy. In operation, **resolutionresidue** negates **<p>** and converts to conjunctive form, stashes the results in a locally bound theory, and invokes the subroutine **rrdisp** on each conjunct. See **rrdisp**, **scheduler**, **tracetask**, **cnf**

resolutionresidues **(resolutionresidues <p>)**
tries to prove the proposition **<p>**. It returns a list of all assumption lists for which it is successful. **Resolutionresidue** is an implementation of linear-input resolution using the set of support control strategy. In operation, **resolutionresidues** negates **<p>** and converts to conjunctive form, stashes the results in a locally bound theory, and invokes the subroutine **rrdisp** on each conjunct. See **rrdisp**, **scheduler**, **tracetask**, **cnf**.

resolutions **(resolutions <p>)**
tries to prove the proposition <p>. It returns a list of all binding lists for which it is successful. **Resolutions** is an implementation of linear-input resolution using the set of support control strategy. In operation, **resolutions** negates <p> and converts to conjunctive form, stashes the results in a locally bound theory, and invokes the subroutine **rsdisp** on each conjunct. See **rsdisp**, **scheduler**, **tracetask**, **cnf**.

rq **(repn <p> rq)**
means that the proposition <p> should be represented in the **req** representation. That is, the **rq-<x>** family of subroutines will be used to retrieve <p>. Its lookup method, **rq-lookup**, takes as its argument a proposition of the form (<r> $<x_1>$ ... $<x_n>$). If there is a LISP subroutine corresponding to the relation symbol <r>, (i.e. on <r>'s **LISP** property,) **rq-lookup** applies the subroutine to the arguments ($<x_1>$ ... $<x_n>$), and returns **truth** if the result is non**nil**; or **nil**. By default, most relations, including arithmetic ones, use this representation. See **re**, **rq-lookup**, **rq-lookups**, **relnproc**, **repn**.

rq-lookup **(rq-lookup <p>)**
is used in trying to retrieve the proposition <p>. See **rq**, **lisp**.

rq-lookups **(rq-lookups <p>)**
is functionally equivalent to **(pluralize (Rq-lookup p))**. See **rq**, **rq-lookup**.

rqb **(repn <p> rqb)**
means that the proposition <p> should be represented in the **rqb** representation. That is, the **rqb-<x>** family of subroutines will be used to retrieve <p>. Its lookup method, **rqb-lookup**, takes as its argument a proposition of the form (<r> $<x_1>$ ... $<x_n>$). If there is a LISP subroutine corresponding to the relation symbol <r>, (i.e. on <r>'s **LISP** property,) **rqb-lookup** applies the subroutine to the arguments ($<x_1>$ ... $<x_n>$), and simply returns the result (assumed here to be a binding-list). See **computable-repn**, **reb**, **rqb-lookup**, **rqb-lookups**, **relnproc**, **repn**.

rqb-lookup **(rqb-lookup <p>)**
is used in trying to retrieve the proposition <p>. See **rqb**, **rqbm**, **lisp**.

rqb-lookups **(rqb-lookups <p>)**
is functionally equivalent to **(pluralize (Rqb-lookup p))**. See **rqb**, **rqb-lookup**.

rqbm **(repn <p> rqbm)**
means that the proposition <p> should be represented in the **rqbm** representation. That is, the **rqbm-<x>** family of subroutines will be used to retrieve <p>. Its lookup method, **rqb-lookup**, takes as its argument a proposition of the form (<r> $<x_1>$ ... $<x_n>$). If there is a LISP subroutine corresponding to the relation symbol <r>, (i.e. on <r>'s **LISP** property,) **rqb-lookup** applies the subroutine to the arguments ($<x_1>$ ... $<x_n>$), and simply returns the result (assumed here to be a list of binding-lists). See **computable-repn**, **rqb**, **rqbm-lookup**, **rqb-lookup**, **relnproc**, **repn**.

rqbm-lookup **(rqbm-lookup <p>)**
is functionally equivalent to **(singularize (Rqb-lookup p))**. See **rqbm**, **rqb-lookup**.

rqfm **(repn <p> rqfm)**
means that the proposition **<p>** should be represented in the **rqfm** representation. That is, the **rqfm-<x>** family of subroutines will be used to retrieve **<p>**. Its lookup method, **rqfm-lookups**, takes as its argument a proposition of the form **(<r> $<x_1>$ ... $<x_n>$ $<x_{n+1}>$)**. It first evaluates the term **(<r> $<x_1>$ ... $<x_n>$)** by applying LISP subroutine corresponding to the function symbol **<r>**, (i.e., on **<r>**'s **LISP** property,) to the list $<x_1>$ ... $<x_n>$. This returns a list of values. **Rqfm-lookups** then unifies each of these with $<x_{n+1}>$, returning the list of results. See **computable-repn**, **rqfm-lookup**, **rqfm-lookups**, **relnproc**, **repn**.

rqfm-lookup **(rqfm-lookup <p>)**
is functionally equivalent to **(singularize (Rqfm-lookups p))**. See **rqfm**, **rqfm-lookups**.

rqfm-lookups **(rqfm-lookups <p>)**
is used to retrieve the propositions **<p>**. See **rqfm**, **lisp**.

rrdisp **(rrdisp <l> <cl> <al> <th>)**
performs one resolution step in trying to derive a contradiction from the propositions on the list **<l>**. The list **<cl>** contains a list of assumptions made in preceding steps. The binding list **<al>** holds the bindings of the variables from preceding steps. The theory **<th>** holds the conjuncts from the negated goal. To be used, all propositions must be in conjunctive form. In addition, only base-level variables are treated as variables, any meta-level variables are treated as constants. Given a goal list **(<p> . <l>)**, **rrdisp** generates subgoals by negating **<p>** to get **<q>** and looking in the data base for propositions of the form **<q>** or **(or ... <q> ...)**. It also uses **trtrueps** to discover whether **p** is assumable. If it is assumable and if it is a ground proposition after plugging in the variable bindings returned by **trtrueps**, **brdisp** creates a new theory that includes **<th>**, asserts the proposition in that theory, and generates appropriate subgoals. The asserted propositions are useful in that they make possible consistency checking before making assumptions in subsequent steps. The order in which multiple **rrdisp** tasks are executed can be influenced via appropriate **preferred** propositions. See **assumable**, **cnf**.

rsdisp **(rsdisp <l> <al> <th>)**
performs one resolution step in trying to derive a contradiction from the propositions on the list **<l>**. The binding list **<al>** holds the bindings of the variables from preceding steps. The theory **<th>** holds the conjuncts from the negated goal. To be used, all propositions must be in conjunctive form. In addition, only base-level variables are treated as variables, any meta-level variable is treated as a constant. Given a goal list **(<p> . <l>)**, **rsdisp** generates subgoals by negating **<p>** to get **<q>** and looking in the data base for propositions of the form **<q>** or **(or ... <q> ...)**. The order in which multiple **rsdisp** tasks are executed can be influenced via appropriate **preferred** propositions. See **cnf**.

runnable **(runnable <k>)**

states that the task **<k>** is **runnable**. A **runnable** task is **applicable** if its operator is a Lisp subroutine; otherwise, it is assumed to be a defined task, and a corresponding invocation of **exec** is **applicable**. In MRS demons are implemented via **runnable**, e.g. to tell the system that it should print a greeting whenever the user asserts a proposition that someone is logged, one simply asserts **(if (loggedin $x) (runnable (print hello t)))** and **(toassert (loggedin &x) fc)**. See **applicable**, **scheduler**.

samep **(samep <x> <y>)**

determines whether expressions **<x>** and **<y>** are the same under consistent variable renaming, and if so returns a binding list for the variables in **<x>**. For example, **(p $x $y $x)** is the same as **(p $y $x $y)** but not **(p $x $y $y)**.

scheduler **(scheduler)**

Scheduler is the heart of the MRS system. It is a simple deliberation-action loop that, at each point in time, decides on an executable task, executes it, and repeats. In its most general state, the choice of task is made by calling **trtruep** to find a task **<k>** such that **(executable <k>)** is true. After the task is performed, the fact is recorded by calling **trassert** on the proposition **(executed <k>)**.

In its initial state, MRS contains a number of propositions to help **scheduler** decide on an executable task. In particular, a task is **executable** if it is **applicable** and is not **disqualified**. Once a task becomes **applicable**, it remains applicable until it is **executed**. One way an applicable task can be **disqualified** is for there to be another applicable task that is **preferred** to it. A **runnable** task is **applicable** if its operator is a Lisp subroutine. Otherwise, it is assumed to be a defined task, and a corresponding **exec** task is **applicable**. CONTINUED ....

scheduler [continued]

This full generality is available only if the switches **executable** and **executed** are both non-nil. For reasons of efficiency, both of these switches are initially set to nil, and an optimized version of this loop is used instead. In particular, the set of **applicable** tasks is kept as the value of the variable **agenda**, and an **executable** task is obtained from this list. If the switch **preferred** is nil, the first element of the list is taken; otherwise, **scheduler** uses **trtruep** to compare the elements using the **preferred** relation. This optimization is fully consistent with the axioms described above. However, it is recommended that the user not change the settings of **executable** and **executed** without careful forethought.

Debugging facilities for the scheduler architecture are not very good at this point. However, rudimentary debugging is possible using **tracetask**.

setdiff **(setdiff <x> <y> <b>)**
means that list **<b>** is all the elements in list **<x>** that are not in list **<y>**.

```
(setdiff nil $y nil)
(if (and (not (element $e $y)) (setdiff $l $y $s))
    (setdiff ($e . $l) $y ($e . $s)))
(if (and (element $e $y) (setdiff $l $y $s))
    (setdiff ($e . $l) $y $s))
```

Procedural attachment: **truep-setdiff**. The lisp file **set** must be loaded from the mrs directory.

setof **(setof <x> <p> <s>)**
means that **<s>** is the set of all objects **<x>** that satisfy **<p>**.

```
(if (and (bagof $x $p $b) (elementsin $b $s))
    (setof $x $p $s))
```

Procedural attachments: **truep-setof** and **lookup-setof**. The lisp file **set** must be loaded from the mrs directory.

singlep **(singlep <p>)**
returns **t** when the proposition **<p>** has at most one solution, i.e. when it is a ground proposition or an atomic proposition with a functional operator and ground arguments. See **function**.

singularize **(singularize <x>)**
returns the singular-value of **<x>**. That is, it returns **(car <x>)**. See **lookupbylookups**. **truepbytrueps**.

stash **(stash <p>)**
stores the proposition **<p>** in the data base. **Stash** is an abstract operator implemented using **kb** and **tostash**.

stash-and **(stash-and (and $<p_1>$ ... $<p_n>$))**
separately stashes each of the conjuncts $<p_1>$, ..., $<p_n>$.

stashapplicable **(stashapplicable (applicable <k>))**
adds **<k>** to **agenda**. See **applicable**.

subclass **(subclass $<c_1>$ $<c_2>$)**
means that the set $<c_2>$ is a subset of $<c_1>$ -- that is, all members of $<c_2>$ are members of $<c_1>$. E.g., **(subclass number integer)**. See **classes**.

subset **(subset <x> <y>)**
means that every element of the list **<x>** is an element of the list **<y>**.

```
(subset nil $y)
(if (and (element $e $y) (subset $l $y))
    (subset ($e . $l) $y))
```

Procedural attachment: **truep-subset**. The lisp file **set** must be loaded from the mrs directory.

succeed **(succeed <z>)**
is a special control form. Executing this form causes the enclosing **doable** task to succeed or the enclosing **undoable** task to fail. In addition, all other subtasks are discarded.

task
In MRS the task of performing an operator **<op>** with arguments $\langle x_1 \rangle$, . . ., $\langle x_n \rangle$ is written **(<op>** $\langle x_1 \rangle$ . . . $\langle x_n \rangle$**)**. The operator in task **<k>** may be a Lisp subroutine, an MRS subroutine defined using **def**, or a special control form like **doand**, **doable**, or **doall**. If the operator is a Lisp subroutine, the task must have have an additional argument for the output value, and it will succeed only if the last argument unifies with the result of calling the subroutine on all but the last argument. For example, the task **(cdr (a b c) (&x . &y))** will succeed with the variable **&x** bound to **b** and the variable **&y** bound to **(c)**. See **execute**, **executes**.

tb **(tb <op> <x1> ... <xn>)**
makes the task **(<op> <x1> ... <xn>)** applicable by placing it on the **agenda**. See **applicable**, **agenda**.

template **(template <x> <t>)**
means that expression **<x>** should be output as **<t>**. In particular, if an expression **<y>** matches **<x>** with binding list **<a1>**, **(output <x>)** returns a copy of **<t>** in which each variable is replaced by the result of calling **output** on its value in **<a1>**. Templates are used by **output**.

terms **(domain <x> <i> terms)**
means that the **<i>**th argument to the subroutine **<x>** should be a term. See **domain**.

thassert **(thassert <p> <th>)**
binds **theory** to **<th>** and asserts the proposition **<p>**. See **assert**, **theory**.

theories **(domain <x> <i> theories)**
means that the **<i>**th argument to the subroutine **<x>** should be a theory. See **domain**.

theory
has as its value the name of the current theory. All propositions stored in the data base via **pr-stash** are associated with the theory named as the value of **theory** at the time of the stash. One can associate a proposition with more than one theory by repeating the call to **pr-stash** with different values for **theory**. Calling **pr-unstash** removes a proposition only from the current theory. The theory named as the value of **theory** is always active, i.e. the propositions associated with it are available for retrieval by **pr-lookup** and **pr-lookups**. See **pr-stash**, **pr-unstash**, **pr-lookup**, and **pr-lookups**.

thfalse **(thfalse (unprovable <p>))**
calls **truep** on the proposition **<p>**. It returns **nil** if the answer is non-nil; otherwise, it returns **truth**.

threpn **(threpn <p> <rpn> <th>)**
means that the representation **<rpn>** should be used to store and access the proposition **<p>** when **<th>** is an active theory. The effect of having conflicting representations for a proposition stored in different theories is undefined when both theories are active. (See **repns** entry for list of allowable representations.) See **activate**, **deactivate**, **theory**, **achieve**, **repns**, **repn**, **repn-assert**, **repn-method**, **repn-unassert**.

thstash **(thstash <p> <th>)**
binds **theory** to **<th>** and stashes the proposition **<p>**. See **stash**, **theory**.

thtrue **(thtrue (provable <p>))**
calls **truep** on the proposition **<p>** and returns the answer.

thunassert **(thunassert <p> <th>)**
binds **theory** to **<th>** and unasserts the proposition **<p>**. See **unassert**, **theory**.

thunstash **(thunstash <p> <th>)**
binds **theory** to **<th>** and unstashes the proposition **<p>**. See **unstash**, **theory**.

tl **(repn <p> tl)**
means that the proposition **<p>** should be represented in the **tl** representation, i.e. the **tl-<x>** family of subroutines will be used to stash, unstash, and lookup **<p>**. This representation is particularly useful for storing propositions involving unary relations, e.g. **(function f)**. Note that propositions stored in this way are not associated with any particular theory and cannot be found by PR-based routines like **prfacts** or **prcontents**. See **repn**, **tl-lookup**, **tl-stash**,**tl-unstash**

tl-lookup **(tl-lookup (<r> <a>)**
returns truth if there is an **<r>** property on the atom **<a>**. Both **<r>** and **<a>** must be atoms. See **tl**.

tl-stash **(tl-stash (<r> <a>))**
sets the **<r>** property of the lisp atom **<a>** to **t**. Both **<r>** and **<a>** must be atoms. See **tl**.

tl-unstash **(tl-unstash (<r> <a>))**
removes the **<r>** property of **<a>**. Both **<r>** and **<a>** must be atoms. See **tl**.

tm-unassert **(tm-unassert <p>)**
calls **unstash** on **<p>** and then calls **unassert** on any proposition all of whose justifications depend on **<p>**. See **just**.

to<g> **(to<g> <p> <f>)**
means that the subroutine **<f>** is to be called in performing the action **<g>** on argument **<p>**. Each of MRS's user-level commands has associated with it a relation that specifies the subroutine to be used in carrying out that command. The relation is named by prefixing the command's name with **to**, e.g. **toAssert** from **assert**. See **kb**.

toachieve **(toachieve <p> <m>)**
means that the method **<m>** should be used to perform the **achieve** action for all propositions which match **<p>**. See **kb**, **achieve**, **to<x>**.

toassert **(toassert <p> <m>)**
means that the method **<m>** should be used to perform the **assert** action for all propositions which match **<p>**. See **kb**, **assert**, **to<x>**.

tocache **(tocache <p> <m>)**
means that the method **<m>** should be used to cache propositions which match **<p>**. (This **<m>** will only be used when the variable **cache** has a non**NIL** value.) See **cache**, **cachebystash**, **to<x>**

tolookup **(tolookup <p> <m>)**
means that the method **<m>** should be used to determined the **lookup** value for all propositions which match **<p>**. See **kb**, **lookup**, **to<x>**, **tolookups**.

tolookups **(tolookups <p> <m>)**
means that the method **<m>** should be used to determined the **lookups** values for all propositions which match **<p>**. See **kb**, **lookups**, **to<x>**, **tolookup**.

toperceive **(toperceive <p> <m>)**
means that the method **<m>** should be used to determined the **perceive** value for all propositions which match **<p>**. See **kb**, **perceive**, **to<x>**, **toperceives**.

toperceives **(toperceives <p> <m>)**
means that the method **<m>** should be used to determined the **perceives** values for all propositions which match **<p>**. See **kb**, **perceives**, **to<x>**, **toperceive**.

toplevel **(toplevel)**
is a read-execute-print loop. See **execute**.

tostash **(tostash <p> <m>)**
means that the method **<m>** should be used to perform the **stash** action for all propositions which match **<p>**. See **kb**, **stash**, **to<x>**.

totruep **(totruep <p> <m>)**
means that the method **<m>** should be used to determined the **truep** value for all propositions which match **<p>**. See **kb**, **truep**, **to<x>**, **totrueps**.

totrueps **(totrueps <p> <m>)**
means that the method **<m>** should be used to determined the **trueps** values for all propositions which match **<p>**. See **kb**, **trueps**, **to<x>**, **totruep**.

tounachieve **(tounachieve <p> <m>)**
means that the method **<m>** should be used to perform the **unachieve** action for all propositions which match **<p>**. See **kb**, **unachieve**, **to<x>**.

tounassert **(tounassert <p> <m>)**
means that the method **<m>** should be used to perform the **unassert** action for all propositions which match **<p>**. See **kb**, **unassert**, **to<x>**.

tounstash **(tounstash <p> <m>)**
means that the method **<m>** should be used to perform the **unstash** action for all propositions which match **<p>**. See **kb**, **unstash**, **to<x>**.

tracetask **(tracetask [<p>])**
As each task is executed, **tasktrace** prints out the name of the subroutine and its arguments provided they match **<p>**. If there is no <p> argument in the subroutine call then a list of all tasks which are to be traced is printed. See **untracetask**.

trassert **(trassert <p>)**
asserts the proposition **<p>** and performs forward chaining as appropriate. **Trassert** is MRS's meta-level assertion routine and is called by many MRS subroutines. The default is simply to stash a proposition, but there are also built-in procedural attachments for propositions containing certain special relations (stored on each relation as the **assert** property). A frequently used procedural attachment is the depth-first forward chaining program **trfc**.

trlookup **(trlookup <p>)**
looks up the proposition **<p>**. If successful, it returns the corresponding binding list; otherwise, it returns **nil**. **Trloookup** is one of MRS's meta-level lookup routines and is called by many MRS subroutines. The default procedure uses indexp and matchp to find any matching propositions in the **pr** representation, but there are also built-in procedural attachments for propositions containing many common relations (stored on each relation as its **lookup** or **lookups** property).

trlookups **(trlookups <p>)**
looks up the proposition **<p>** and returns a binding list for each matching proposition that it finds. **Trlookups** is one of MRS's meta-level lookup routines and is called by many MRS subroutines. The default procedure uses indexp and matchp to find any matching propositions in the **pr** representation, but there are also built-in procedural attachments for propositions containing many common relations (stored on each relation as its **lookup** **lookups** property).

trstash **(trstash <p>)**
stashes the proposition **<p>**. **Trstash** is MRS's meta-level stash routine and is called by many MRS subroutines. The default is **pr-stash**, but there are also built-in procedural attachments for propositions containing many common relations (stored on each relation as its **stash** property).

trtruep **(trtruep <p>)**
tries to prove the proposition **<p>**. If successful, it returns a corresponding binding list; otherwise, it returns **nil**. **Trtruep** is one of MRS's meta-level theorem proving routines and is called by many MRS subroutines. Only meta-level variables are treated as variables by **trtruep**, and all base-level variables are treated as constants. The inference procedure used is the depth-first backward chaining program **trbc**, but there are also built-in procedural attachments for propositions containing many common relations (stored on each relation as its **truep** or **trueps** property).

trtrueps **(trtrueps <p>)**
tries to prove the proposition **<p>** and returns a list of all binding lists for which it is successful. **Trtrueps** is one of MRS's meta-level theorem proving routines and is called by many MRS subroutines. Only meta-level variables are treated as variables by **trtrueps**, and all base-level variables are treated as constants. The inference procedure used is the depth-first backward chaining program **trbcs**, but there are also built-in procedural attachments for propositions containing many common relations (stored on each relation as its **truep** or **trueps** property).

truep **(truep <p>)**
tries to prove the proposition **<p>**. If it is successful, it returns a binding list for the base-level variables in **<p>**; otherwise, it returns **nil**. **Truep** is an abstract operator implemented using **kb** and **totruep**.

truep-bagof **(truep-bagof (bagof <x> <p> <s>))**
calls **trueps** on **<p>** and matches **<s>** against the list formed by plugging the answers into **<x>**.

truep-is **(truep-is (is <x> <y>))**
uses **getval** to evaluate the arbitrarily nested expression **<x>** and tries to unify the answer with **<y>**. See **is**.

truepbytrueps **(truepbytrueps <p>)**
is equivalent to **(singularize (trueps <p>))**.

trueps **(trueps <p>)**
tries to prove the proposition **<p>** and returns a list of all binding lists for which it is successful. **Trueps** is an abstract operator implemented using **kb** and **totrueps**.

truepsbytruep **(truepsbytruep <p>)**
is equivalent to **(pluralize (truep <p>))**.

trunassert **(trunassert <p>)**
unasserts the proposition **<p>**. **Trunassert** is MRS's meta-level unassertion routine and is called by many MRS subroutines. The default is simply to unstash a proposition, but there are also built-in procedural attachments for propositions containing certain special relations (stored on each relation as the **unassert** property).

trunstash **(trunstash <p>)**
unstashes the proposition **<p>**. **Trunstash** is MRS's meta-level unstash routine and is called by many MRS subroutines. The default is **pr-unstash**, but there are also built-in procedural attachments for propositions containing many common relations (stored on each relation as its **unstash** property).

truth
has **((t . t))** as its value. The value of truth occurs as the last pair in binding lists returned by MRS's retrieval and inference procedures.

tutor **(tutor)**
runs an interactive tutor that introduces one to the basic representation and inference mechanisms of MRS.

unassert **(unassert <p>)**
removes the proposition **<p>** from the data base and performs all appropriate inference. **Unassert** is an abstract operator implemented using **kb** and **tounassert**.

unassert-and **(unassert-and (and $<p_1>$ ... $<p_n>$))**
separately unasserts each of the conjuncts $<p_1>, \ldots, <p_n>$.

unassert-iff **(unassert-iff (if <p> <q>))**
asserts **(if <p> <q>)** and **(if <q> <p>)**.

undoable **(undoable <k>)**
designates the task of trying to execute the task **<k>**. The task **(undoable <k>)** succeeds only if there is no successful execution of **<k>**. Note that as a result of the current implementation, it is not possible to interleave subtasks outside a **doable** task with those inside. See **succeed** and **cut**.

unifyp **(unifyp <x> <y>)**
determines whether expressions **<x>** and **<y>** are unifiable, and if so returns their most general unifier. **Unifyp** differs from **matchp** in that multiple occurrences of the same variable in both **<x>** and **<y>** are not treated as distinct variables. For example, **(p $x b)** and **(p a $x)** are not unifiable, but they do match.

unincludes **(unincludes $<t_1>$ $<t_2>$)**
removes any **includes** link between theories $<t_1>$ and $<t_2>$. See **includes**.

union **(union <x> <y> <b>)**
means that list **<b>** is the lists **<x>** and **<y>** appended together.

```
(union nil $y $y)
(if (union $l $y $s)
    (union ($x . $l) $y ($x . $s)))
```

Procedural attachment: **truep-union**. The lisp file **set** must be loaded from the mrs directory.

unstash **(unstash <p>)**
removes the proposition **<p>** from the data base. Note this is not equivalent to asserting the negation of **<p>**. **Unstash** is an abstract operator implemented using **kb** and **tounstash**.

unstash-and **(unstash-and (and <p_1> ... <p_n>))**
separately unstashes each of the conjuncts **<p_1>**, . . ., **<p_n>**.

unstashapplicable **(unstashapplicable (applicable <k>))**
removes **<k>** from **agenda**. See **applicable**.

untracetask **(untracetask [<p>])**
untraces the task **<p>**. If there is no **<p>** argument in the subroutine call then it untraces all **<p>** that are currently being traced. See **tracetask**.

value **(value <x> <y>)**
means that the atom **<x>** has value **<y>**.

variable **(variable <x>)**
means that the symbol **<x>** is a variable.

varp **(varp <xp>)**
returns a non-nil value if **<xp>** is a variable and otherwise returns nil. See **blvarp** and **mlvarp**.

where **(where <p>)**
prints out a message for each recorded justification in which **<p>** is a premise. The message includes information about the justified proposition, the inference method, and all premises. See **justify**, **just**.

why **(why <p>)**
prints out a message for each recorded justification for the proposition **<p>**. The message includes information about the relevant inference method and all premises. See **justify**, **just**.

Appendix I
Base-Level Vocabulary

Appendix II
Meta-Level Vocabulary

Appendix III
Subroutines

Appendix IV
Variables

Appendix V
Concepts

Index