Table of Contents

Page

1 Overview of Advanced Joshua Concepts **1**

1.1 The Joshua Protocol of Inference 2
1.2 The Default Implementation of the Protocol 2
1.3 Customizing the Joshua Protocol 5

2 Storing and Retrieving Knowledge in Joshua: the Virtual Database **7**

2.1 What is a Virtual Database? 7
2.2 Predications as Instances 7
2.3 The Joshua Database Protocol 8
2.3.1 The Contract of the Generic Function **joshua:insert** 9
2.3.2 The Contract of the Generic Functions **joshua:ask-data** and **joshua:fetch** 10
2.3.3 The Contract of the Generic Function **joshua:uninsert** 14
2.3.4 The Contract of the Generic Function **joshua:clear** 14
2.4 Joshua's Default Database: the Discrimination Net 16
2.4.1 Organization of the Default Discrimination Net 17

3 The Joshua Rule Facilities **23**

3.1 Advanced Features of Joshua Rules 24
3.2 The Joshua Rule Compiler 26
3.2.1 The Forward Rule Compiler 27
3.2.2 The Backward Rule Compiler 33
3.3 Ordering Rule Execution 35
3.4 Controlling Rule Invocation 35
3.5 The Joshua Rule Indexing Protocol 36
3.5.1 The Contract of the Trigger Adding Functions 38
3.5.2 The Contract of the Trigger Deleting Functions 38
3.5.3 The Contract of the Trigger Locating Functions 39
3.5.4 The Contract of the Trigger Mapping Functions 41

4 The Joshua Question Facilities **47**

4.1 Controlling Question Invocation 47
4.2 The Joshua Question Indexing Protocol 48
4.2.1 The Contract of **joshua:add-backward-question-trigger** 48
4.2.2 The Contract of **joshua:delete-backward-question-trigger** 48
4.2.3 The Contract of **joshua:locate-backward-question-trigger** 49
4.2.4 The Contract of **joshua:map-over-backward-question-triggers** 50

5 Truth Maintenance Facilities **53**
5.1 The Truth Maintenance Protocol 54
5.1.1 The Contract of the Joshua TMS Protocol Functions 54
5.1.2 The Contract of a Joshua TMS Justification 55
5.1.3 TMS Utility Routines 56
5.1.4 Signalling Contradictions and Managing Backtracking 57
5.1.5 Signalling Truth Value Changes 63
5.2 The Joshua LTMS 65
5.2.1 Clause Justification Structures 65

6 Joshua Metering **73**
6.1 Joshua Metering Types 73
6.1.1 Joshua Tell Metering 73
6.1.2 Joshua Ask Metering 75
6.1.3 Joshua Merge Metering 76
6.2 Choosing Joshua Metering Types 77

7 Controlling Data and Rule Indexing **79**
7.1 Customizing the Data Index 81
7.1.1 Customizing the Data Index Without Storing Predications 85
7.2 Customizing the Rule Index 88
7.3 Customizing the Rule Compiler 92
7.3.1 Customizing the Matchers Generated by the Rule Compiler 102

8 The Joshua Object Facility **105**
8.1 Introduction to the Joshua Object Facility 105
8.2 Basic Capabilities of the Joshua Object Facility 107
8.3 Using Paths to Refer to the Structure of an Object 109
8.4 Type Hierarchy in the Joshua Object Facility 110
8.5 Part-Whole Hierarchy in the Joshua Object Facility 112
8.6 Other Capabilities of Slots 113
8.6.1 Initial Values of Slots 113
8.6.2 Set Valued and Single Valued Slots 113
8.6.3 Slots and Truth Maintenance 114
8.6.4 Slots and Attached Actions 114
8.6.5 Invoking Methods Associated with the Object Associated with a Slot 115
8.6.6 Equalities Between Slot Values 116
8.7 Other Options in Define-Object-Type 117
8.8 The Predicates Used in the Joshua Object Facility 118

9 Joshua Language Dictionary **121**
9.1 Dictionary Entries 121

List of Figures

Page
1 The Joshua Protocol of Inference 4
2 Type Network for non-TMS Predicates 4
3 Type Network for LTMS Predicates 4
4 The Default Implementation of the Protocol of Inference 5
5 The **tell** data-indexing protocol and its default implementation 11
6 The **ask-data** protocol and its default implementation 13
7 The **untell** protocol and its default implementation 15
8 The **clear** protocol and its default implementation 16
9 Sample Discrimination Net Display 18
10 Summary of Joshua Rule Operation 23
11 Sample Rete Network 29
12 Sample Rete Network Display 30
13 Sample Rete Network Display with Filter Nodes 31
14 Sample Rete Network Display with **or** Node 32
15 Rete Network For Rule with Nested Ands 32
16 Rule Indexing Protocol 38
17 The Trigger-Adding Protocol and Default Implementation 39
18 The Trigger-Deleting Protocol and Default Implementation 40
19 The **justify** protocol and its default implementation 42
20 The ask-**rules** protocol and its default Implementation 44
21 The Question Protocol 48
22 The Question Trigger Adding Protocol and Default Implementation 49
23 The Question Trigger Deleting Protocol and Default Implementation 49
24 The **ask-questions** protocol and its default implementation 51
25 Example Trace of Condition Handler 64
26 Example of setting up a nogood clause 66
27 Tell metering of the unmodelled good-to-eat predicate. 75
28 Tell metering of the modelled good-to-eat predicate. 76
29 Knowledge Structures Can Be Diversely Implemented 79
30 Graph of the Mixed Chaining Rule Foo 95
31 Trace of The Mixed Chaining Rule Foo 96
32 Graph of Mixed Chaining Rule Foo 98
33 Trace of Explicitly Controlled Mixed Chaining 99
34 Trace of Explicitly Controlled Mixed Chaining 99
35 Predications Being Mapped into an Object Representation 105
36 Other Capabilities of the Object Facility 106
37 A Resistor and its Representation as an Object 107
38 The Object-Type Hierarchy of Two-Terminal Devices 111
39 Equality Links in a Two Resistor Voltage Divider 117
40 Graph of the Mixed Chaining Rule Foo 174

41 Trace of The Mixed Chaining Rule Foo 174
42 Graph of Mixed Chaining Rule Foo 176
43 Trace of Explicitly Controlled Mixed Chaining 177
44 Trace of Explicitly Controlled Mixed Chaining 177

1. Overview of Advanced Joshua Concepts

Joshua is an extensible software product for building and delivering expert system applications. It is implemented on the Symbolics 3600 and Ivory families, on top of the Symbolics Genera environment. Joshua is optimized for applications where performance and delivered functionality are important.

User's Guide to Basic Joshua, the first manual in the Joshua documentation set, gives an introduction to the Joshua language and development environment. It covers everything you need to know to program using Joshua's built-in facilities.

Among Joshua's strengths is that this system is a coherent, multi-level environment, making advanced features available when you need them. Joshua is built around some 30 core functions, the Protocol of Inference, which are *accessible to the user for modification*.

This modularity and accessibility offer powerful advanced features: user interfaces, control structures, storage structures can all be customized to reflect what is most natural for the application; external databases can be accessed; existing software tools can be seamlessly integrated into the Joshua application; performance can be fine-tuned.

This documentation volume, *Joshua Reference Manual*, describes in detail the protocol of inference and the default implementation of that protocol supplied as part of the Joshua system. In addition, it describes how you can customize Joshua to your own particular application. We often refer to this tailoring or customization process as *modeling*.

The specific topics covered here include:

- The Database Protocol
- The Default Discrimination Net
- The Rule Compiler
- The Rule Indexing Protocol
- The Question Indexing Protocol
- The Truth Maintenance Facilities
- The Joshua LTMS
- Controlling Data and Rule Indexing

The implementation of the Joshua protocol of inference depends heavily on the object-oriented programming facilities of Symbolics Common Lisp. These same features will be included in the Common Lisp Object System. A working knowledge of

the concepts of this style of object-oriented programming will be helpful in understanding how to customize the protocol of inference. For more information, see the section "Flavors" in *Symbolics Common Lisp Programming Constructs*.

1.1. The Joshua Protocol of Inference

Each different Joshua predicate is implemented as an object type (*flavor* in Symbolics Common Lisp terminology, *class* in the Common Lisp Object System). Each protocol step is implemented as a generic function, so that generic dispatch can select the method appropriate for that function and that predicate. By defining your own methods for protocol functions for particular predicates or groups of predicates, you customize Joshua's behavior for those functions and predicates.

The protocol of inference is a way of grouping the many steps of the inferencing process into a functional hierarchy. Figure1 shows the hierarchy of generic functions.

This grouping of the protocol functions splits the protocol into relatively independent parts. For example, an implementation of the TMS protocol should work with just about any implementation of the database interface. This independence enhances sharing of code between different applications, and makes the whole protocol easier to understand.

The protocol imposes a level of modularity on your application which will help you organize your program and think about its many parts in a more coherent way. Conversely, the many levels of the protocol allow you to customize the protocol with "just the right amount" of effort. Although comprehensive changes may require significant effort, simple changes require minimal effort. In all cases, the careful organization and definition of the protocol will make your applications easier to design, build, and understand.

1.2. The Default Implementation of the Protocol

The Joshua system provides a complete implementation of the protocol of inference. We refer to this as the *default implementation*, to encourage customization of the protocol. We expect that the default implementation will be perfectly adequate for prototyping and for large parts of production-quality applications. Where the default implementation is lacking, either in features or performance, customization can be done. The fine-grained control offered by Joshua allows this customization to be applied where necessary, while the rest of an application can continue to use the default implementation.

The default implementation of the protocol of inference is provided by a set of object types (flavors in the current implementation) which have methods defined for all the generic functions of the protocol. The object types are arranged so that they may be used either by the default implementation or by user-defined implementations. Figure2 shows the network of types used by the default implementation of non-TMS predicates, **joshua:default-predicate-model**.

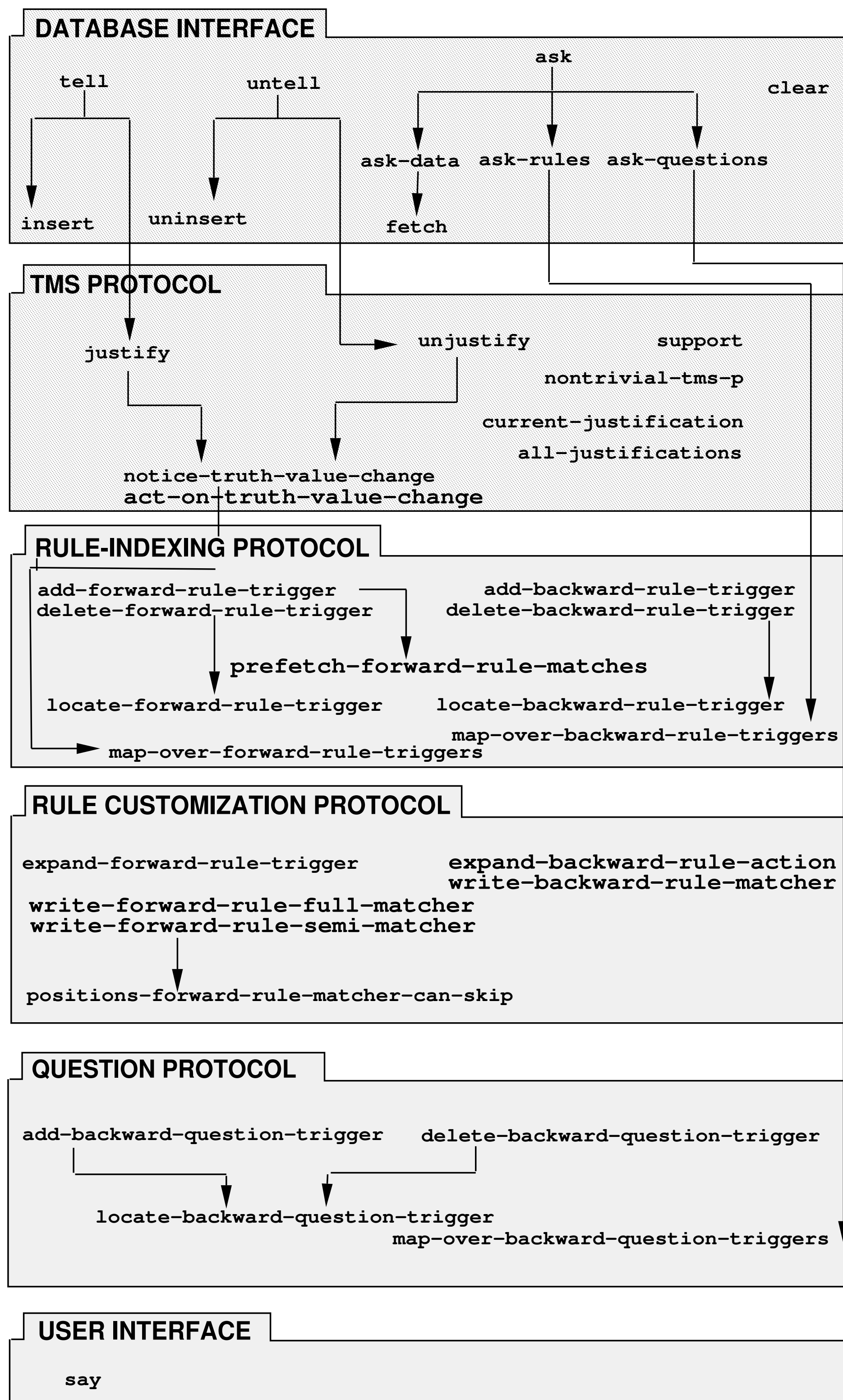

DATABASE INTERFACE
tell
untell
ask
clear
ask-data
ask-rules
ask-questions
insert
uninsert
fetch
TMS PROTOCOL
justify
unjustify
support
nontrivial-tms-p
current-justification
all-justifications
notice-truth-value-change
act-on-truth-value-change
RULE-INDEXING PROTOCOL
add-forward-rule-trigger
delete-forward-rule-trigger
add-backward-rule-trigger
delete-backward-rule-trigger
prefetch-forward-rule-matches
locate-forward-rule-trigger
locate-backward-rule-trigger
map-over-backward-rule-triggers
map-over-forward-rule-triggers
RULE CUSTOMIZATION PROTOCOL
expand-forward-rule-trigger
expand-backward-rule-action
write-backward-rule-matcher
write-forward-rule-full-matcher
write-forward-rule-semi-matcher
positions-forward-rule-matcher-can-skip
QUESTION PROTOCOL
add-backward-question-trigger
delete-backward-question-trigger
locate-backward-question-trigger
map-over-backward-question-triggers
USER INTERFACE
say

Figure 1. The Joshua Protocol of Inference

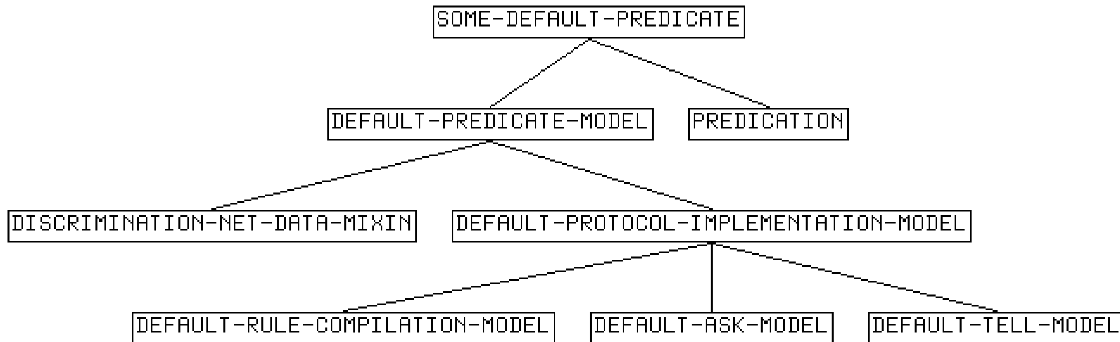


Figure 2. Type Network for non-TMS Predicates

Figure 3 shows the network of types for the LTMS implementation, **ltms:ltms-predicate-model**.

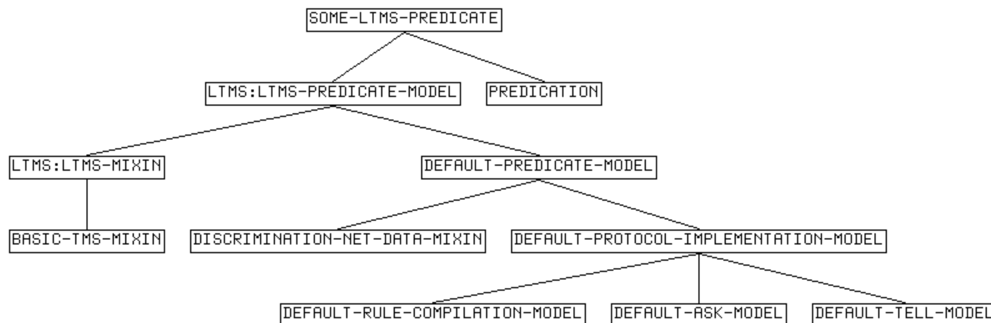


Figure 3. Type Network for LTMS Predicates

Notice that it is built by adding **ltms:ltms-mixin** to **joshua:default-predicate-model**, and so includes as a subgraph all the parts of **joshua:default-predicate-model**. So the basic predicate behavior of LTMS predicates in the default LTMS model comes from **joshua:default-predicate-model**, and **ltms:ltms-mixin** provides the TMS behavior.

Figure 4 shows which methods are associated with each component of the implementation object types.

The implementation techniques chosen for the default should be efficient over a wide range of Joshua programs and applications. These techniques are robust and general. Particular attention has been paid to optimizing them for "typical" applications, and they should prove sufficient for most Joshua programmers' needs. In addition, the default implementation has been optimized for the Symbolics Common Lisp and Genera environment.

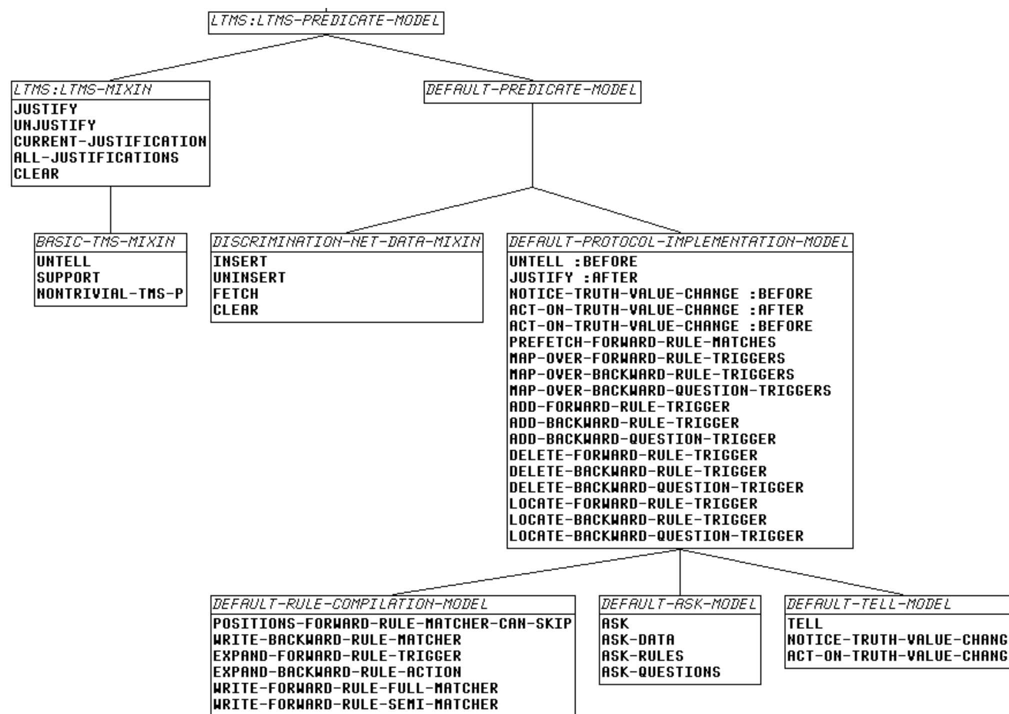


Figure 4. The Default Implementation of the Protocol of Inference

1.3. Customizing the Joshua Protocol

When the default implementation of the protocol of inference is lacking, whether in features or performance, you should customize the protocol.

Since each step of the protocol of inference is implemented as a generic function, you can define your own methods for these functions. In this way you can modify the behavior of Joshua. Each protocol function has a *contract*, or set of things it must do. As long as the contract is followed, the Joshua system will function correctly. The default implementation supplies methods which implement each protocol function correctly. The default techniques have been chosen to be robust, general, and efficient. However, for any particular problem there may be more efficient ways to implement parts of that problem.

We will describe each grouping of protocol functions to show the different ways that the protocol can be customized. An important feature of the protocol is the multi-level nature of the generic function tree. This allows fine-grained control over the customization, so that you can specify as much or as little of the behavior as you need. If you define methods for high-level functions, you are taking over most or all of the behavior. If you wish to change the behavior in less drastic ways, you would define methods for lower-level functions. Descriptions and examples for each part of the protocol will explain the levels and how the different parts of the protocol interact.

2. Storing and Retrieving Knowledge in Joshua: the Virtual Database

2.1. What is a Virtual Database?

Conceptually, a database is an infinitely extensible collection of facts. In Joshua, a database is a structure where you store statements together with associated information, such as truth values. The data is in the form of predications.

The Joshua database protocol makes a *virtual database* possible. That is, the protocol gives you the capability to implement your data structures in any way suitable to your needs; in fact, since different data structures can coexist, you can choose the best data representation for each individual problem piece. This flexibility means you can minimize storage and lookup time for particular kinds of data, thereby increasing the efficiency of your application.

The Joshua database protocol consists of five database generic functions, **joshua:insert**, **joshua:ask-data**, **joshua:fetch**, **joshua:uninsert**, and **joshua:clear**, that are separated from the database implementation functions. This modular organization provides for a stable, consistent interface to diversely implemented data structures.

Joshua's default database is implemented as a *discrimination net*. This is a general-purpose data structure, commonly used in AI, that is reasonably efficient over a wide range of applications. However, for a fixed problem, you can usually do better.

This chapter discusses the general contract of the five data-indexing functions, as well as their default implementation. We also cover the organization of the discrimination net.

2.2. Predications as Instances

Predications have a dual role in Joshua. They store data, and thus are a knowledge representation, that is they "mean" something; they also have program actions associated with them, and in that sense they "do" something. Predications can be remembered, asked about, printed, and so on, as specified by the generic functions in the Joshua protocol.

Although predications look like lists with square brackets, they are really *instances* and each of the operations you perform on them is a generic function. (Readtables change bracketed input to appropriate **joshua:make-predication** forms; print methods arrange for predications to be printed with brackets. But underneath the user interface, predications are just instances.) This lets Joshua keep interface and implementation separate in dealing with predicates, in the same way the Flavor system separates interface (generic functions) from implementation (methods). You use **joshua:define-predicate** to specify the implementation for a given predicate by mixing in all its base models (or flavors).

2.3. The Joshua Database Protocol

Recall that the interface to the Joshua database is controlled by the four protocol generic functions, **joshua:tell**, **joshua:ask**, **joshua:untell**, and **joshua:clear**.

joshua:tell Inserts predication objects (predications and related information) into the database.

joshua:ask Retrieves these predication objects from the database.

joshua:untell Removes a predication object from the database.

joshua:clear Flushes the database.

Each of these four functions dispatches to a method that calls on other generic functions to do part of its work. The generic functions that manage the data indexing are:

joshua:insert Does data indexing for **joshua:tell**. Puts a predication where **joshua:fetch** can find it.

joshua:ask-data Performs unification and calls the continuation on objects retrieved by **joshua:fetch**. If the database does not actually retain the predication you **joshua:tell**, **joshua:ask-data** is the place where one should be reconstructed.

joshua:fetch Does data indexing for **joshua:ask**. Finds a predication object in the place that **joshua:insert** put it. **joshua:fetch** always calls its continuation on a predication that was found in the database.

joshua:uninsert Does data indexing for **joshua:untell**. Removes a predication object from the place that **joshua:insert** put it.

joshua:clear The **joshua:clear** method takes care of data flushing, that is, of resetting the database so that it is completely empty.

The Protocol lets you change the way predications are stored in the virtual database. The section "Customizing the Data Index" covers this topic.

The point to note here is that if you customize your database you must always include methods for all five (or sometimes four) generic functions, namely, **joshua:insert**, **joshua:fetch** or **joshua:ask-data**, **joshua:uninsert**, and **joshua:clear**. This is because they must be consistent in their functionality; **joshua:tell** must know where to put data, **joshua:ask** and **joshua:untell** must know where to find data, and **joshua:clear** must know how to flush data. (It is not always necessary to write a new method for **joshua:ask-data**, since it relies on **joshua:fetch** for database access. Similarly, **joshua:ask-data** when customized, might never call **joshua:fetch**.)

joshua:insert, **joshua:fetch**, **joshua:ask-data**, **joshua:uninsert**, and **joshua:clear** dispatch to the appropriate method for the model the predicate is built on. The default method for **joshua:ask** is on **default-ask-model**. The default model for predi-

cations is **joshua:discrimination-net-data-mixin**, which implements the generic database as a discrimination net. (Note that you would seldom call **joshua:insert** or **joshua:uninsert** directly, except when debugging a data model.)

The general contract of **joshua:insert**, **joshua:ask-data**, **joshua:fetch**, **joshua:uninsert**, and **joshua:clear**, as distinct from their particular implementation, is detailed in the following sections:"The Contract of the Generic Function **joshua:insert**", "The Contract of the Generic Functions **joshua:ask-data** and **joshua:fetch**", "The Contract of the Generic Function **joshua:uninsert**", "The Contract of the Generic Function **joshua:clear**".

For an example of how these functions work together: See the section "Customizing the Data Index", page 81.

2.3.1. The Contract of the Generic Function joshua:insert

joshua:insert stores predication objects in the database, or at least records enough data from which **joshua:ask-data** can reconstruct these predication objects. This function does not deal with the other operations of **joshua:tell**, namely, justification and locating forward rules. These are the responsibility of **joshua:justify**, **joshua:map-over-forward-rule-triggers**, and **joshua:notice-truth-value-change**. See the section "The Joshua Rule Facilities ", page 23. By modularizing the operations of **joshua:tell**, we let you pinpoint the specific functionality you might want to modify; for instance, you can still use the existing **joshua:insert** function, even if you define your own way of doing justification and locating forward rules. (If you want to redefine justification, forward rule mapping, *and* data indexing, all at once, you would, probably, want to redefine the function **joshua:tell** itself. But in almost all cases it is sufficient to move down a level and rewrite only the piece of functionality you need.)

Although you can redefine the database structure, **joshua:insert** always expects data in the form of predications. Once installed by **joshua:tell**, predications are objects containing state information such as justifications. The system usually expects to deal with these objects, not with copies or patterns; for example, the continuation of **joshua:ask** is called with an argument which contains the actual predication object retrieved from the database.

joshua:insert must return two values. If the predication is being added for the first time, **joshua:insert** returns it, as well as the value **joshua::t**.

If a variant of the predication already exists in the database, **joshua:insert** returns the canonical version of it (the version already inserted in the database), together with the value **joshua::nil**.

joshua:insert uses the **joshua:variant** test to determine if the predication it is inserting already exists in the database. Patterns p1 and p2 are variants under the following conditions:

- If the constants in p1 are **joshua::eql** to the constants in p2.

- If the variables in p1 and p2 are in the same places, and if there is a renaming of variables that makes them the same.
- Recursive structures (such as lists and predications) inside a predication must be recursively variants.

For more detail: See the function **joshua:variant**, page 252.

Figure 5 shows the organization of the **joshua:tell** data-inserting protocol including the default implementation of **joshua:insert**.

2.3.2. The Contract of the Generic Functions joshua:ask-data and joshua:fetch

Like those of **joshua:tell**, the operations of **joshua:ask** are modularized to allow fine-tuning of functionality changes. The data-indexing functionality of **joshua:ask** is also broken down into separate functionality assumed by **joshua:ask-data** and **joshua:fetch**.

The contract of **joshua:ask-data** is to do unification on the objects passed to it by **joshua:fetch**, and to call the **joshua:ask** continuation on the unified query and its support.

joshua:ask-data does not deal with backward rules or questions; these are the respective responsibility of **joshua:ask-rules**, and **joshua:ask-questions**, which in turn pass off to **joshua:map-over-backward-rule-triggers** and **joshua:map-over-backward-question-triggers**.

joshua:ask-data is not required to find the canonical predication in the database. That is the responsibility of **joshua:fetch**. If the data model does not store the actual predication, but rather information from which a copy of the predication may be reconstructed, **joshua:ask-data** is the place where this reconstruction should be done.

Loosely defined, the contract of **joshua:fetch** is to get a superset of objects that might unify with the query (including those objects matching the pattern it is given). Note that while **joshua:fetch** deals with its input objects as *patterns* that must be matched, the continuation must be called on predication objects found in the database.

joshua:fetch does not check truth values of **joshua:*true*** or **joshua:*false***; **joshua:ask-data**, on the other hand, does.

joshua:fetch is not required to do unification, as that is the responsibility of **joshua:ask-data**; the contract of **joshua:fetch** merely specifies that it do whatever is convenient at the database level. Thus **joshua:fetch** can fetch anything that might unify with its pattern, skipping only definite failures. For some examples of this: See the section "Organization of the Default Discrimination Net", page 17.

How and to what extent **joshua:fetch** filters objects is up to the implementation. Since filtering is cheap and unification is expensive, the more filtering you can do, the better.

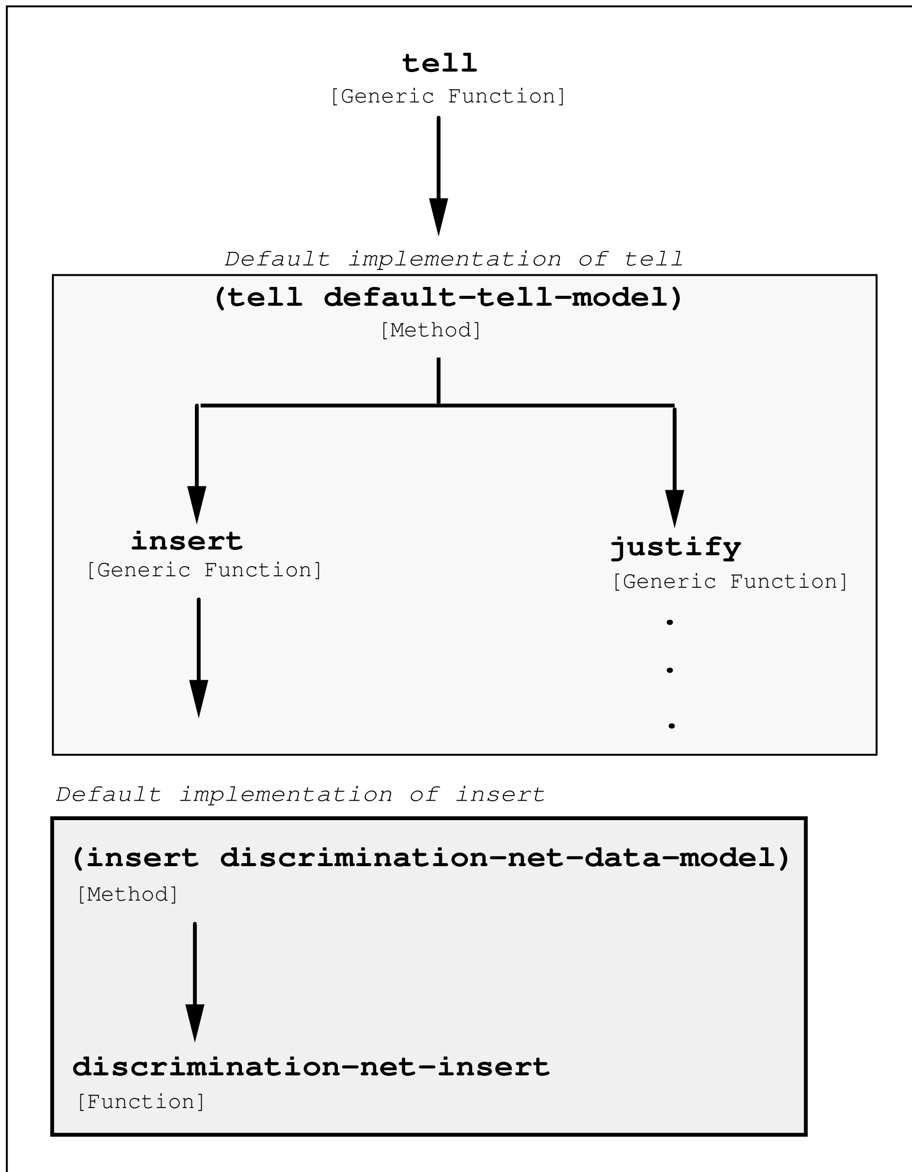


Figure 5. The **tell** data-indexing protocol and its default implementation

One proper, but slow, implementation of **joshua:fetch** is to call the continuation on *every* predication in the database, and let unification do the filtering. That

would be corect, but slow (like a database without indices). Here's an example.

```
(defvar *slow-database* nil "Just a list of all the facts.")

(define-predicate-model slow-data-model () ())

(define-predicate-method (insert slow-data-model) ()
  ;; if this is new data, push it onto the list.
  ;; Otherwise return the canonical version.
  (let ((found (find self *slow-database* :test #'variant)))
    (if found
        (values found nil)
        (progn (push self *slow-database*)
               (values self t)))))

(define-predicate-method (fetch slow-data-model) (continuation)
  ;; indiscriminately suggest every fact as a candidate
  (mapc continuation *slow-database*))

(define-predicate-method (clear slow-data-model) (clear-data-p ignore)
  ;; clearing the database is just setting it to nil
  (when clear-data-p
    (setq *slow-database* nil)))

(define-predicate-method (uninsert slow-data-model) ()
  ;; uninsert just deletes self from the list
  (setq *slow-database* (delete self *slow-database*)))

(compile-flavor-methods slow-data-model)

(define-predicate slow (arg1 arg2) (slow-data-model default-predicate-model))
```

The default implementation of **joshua:fetch** uses the discrimination net. See the section "Joshua's Default Database: the Discrimination Net", page 16.

Figure

6, shows the organization of the **joshua:ask** data-retrieval protocol including the default implementation of **joshua:ask-data** and **joshua:fetch**.

2.3.2.1. Signalling a Condition When joshua:ask-data or joshua:fetch Can't Handle a Query

The Joshua Database Protocol allows you to structure your data in ways that are appropriate for your application; sometimes this involves trading off generality for performance. For example, if a significant portion of your data consists of object-attribute-value triples (such as the *color* of the *block* is *blue*), then you might want to use an object-oriented representation (such as **joshua::flavor** instances) to store this data. However, using this representation makes it awkward or slow to respond to a query that asks for every object with a specific property, such as:

```
[has-eye-color ?who blue]
```

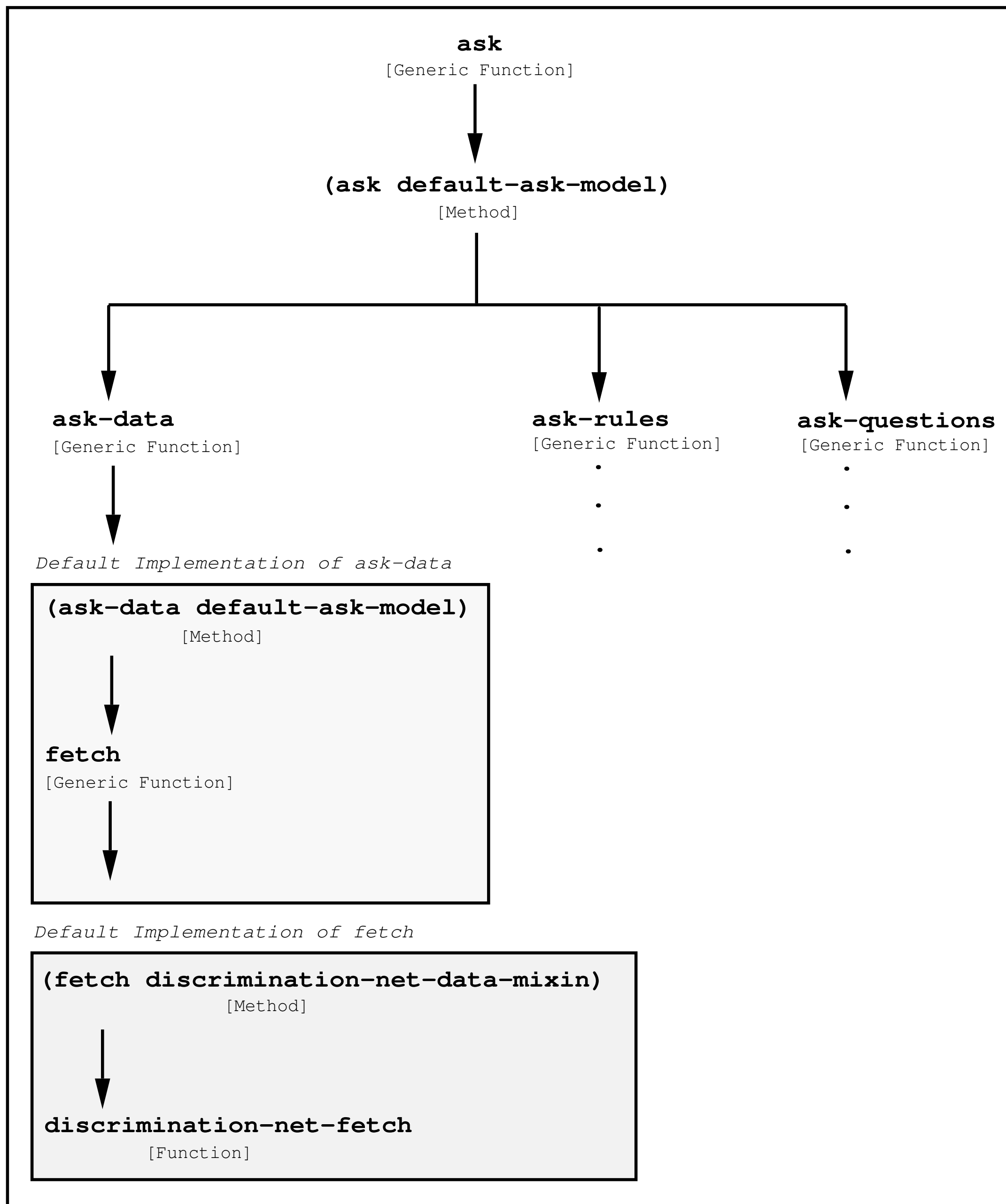


Figure 6. The **ask-data** protocol and its default implementation

An implementation of **joshua:ask-data** or **joshua:fetch** would ideally answer such a query even if it did so slowly. However, such queries may be of such little value to an application that a developer decides not to waste effort on implementing a method that can respond to the query.

It is important, however, that **joshua:fetch** and **joshua:ask-data** methods do not cause errors when faced with a query that they do not wish to handle. One reason

for this is that the command Show Joshua Database may post such a query even if the application never makes such queries on its own.

The contract of **joshua:ask-data** and **joshua:fetch** requires these methods to **joshua::signal** a specific condition when they decline to handle a query. The base flavor for such condition objects is **ji:model-cant-handle-query**. A second condition flavor (built on this base flavor) is called **ji:model-can-only-handle-positive-queries** which (as the name suggests) should be used if the implementation is presented with a negated query, but only expects queries which are not negated.

The following is an example of how to use these conditions:

```
(define-predicate-method (ask-data object-model)
                         (truth-value continuation)
  (unless  (eql truth-value *true*)
    (signal 'ji:model-can-only-handle-positive-queries
            :query self
            :model 'port-direction-model))
  (with-statement-destructured (object value) ()
    (typecase object
      (unbound-logic-variable
        (signal 'ji:model-cant-handle-query
                :model 'port-direction-model
                :query self))
      (otherwise  < whatever you really want to do > ))))
```

2.3.3. The Contract of the Generic Function joshua:uninsert

The contract of **joshua:uninsert** is to remove a single predication object that **joshua:insert** stored into a particular model. **joshua:untell** passes all TMS issues to **joshua:unjustify**. The clearing of internal caches (such as the Rete net), is handled automatically, even if you supply your own method for **joshua:uninsert**.

Figure 7, shows the organization of the **joshua:untell** data removal protocol including the default implementation of **joshua:uninsert**.

2.3.4. The Contract of the Generic Function joshua:clear

The contract of **joshua:clear** is to remove all facts that **joshua:insert** stored into a particular model. Note that if you write a model that redefines **joshua:insert** and **joshua:fetch**, you almost certainly need to write (or inherit) a corresponding **joshua:clear** method.

Figure 8 shows the organization of the database clearing protocol, including its default implementation.

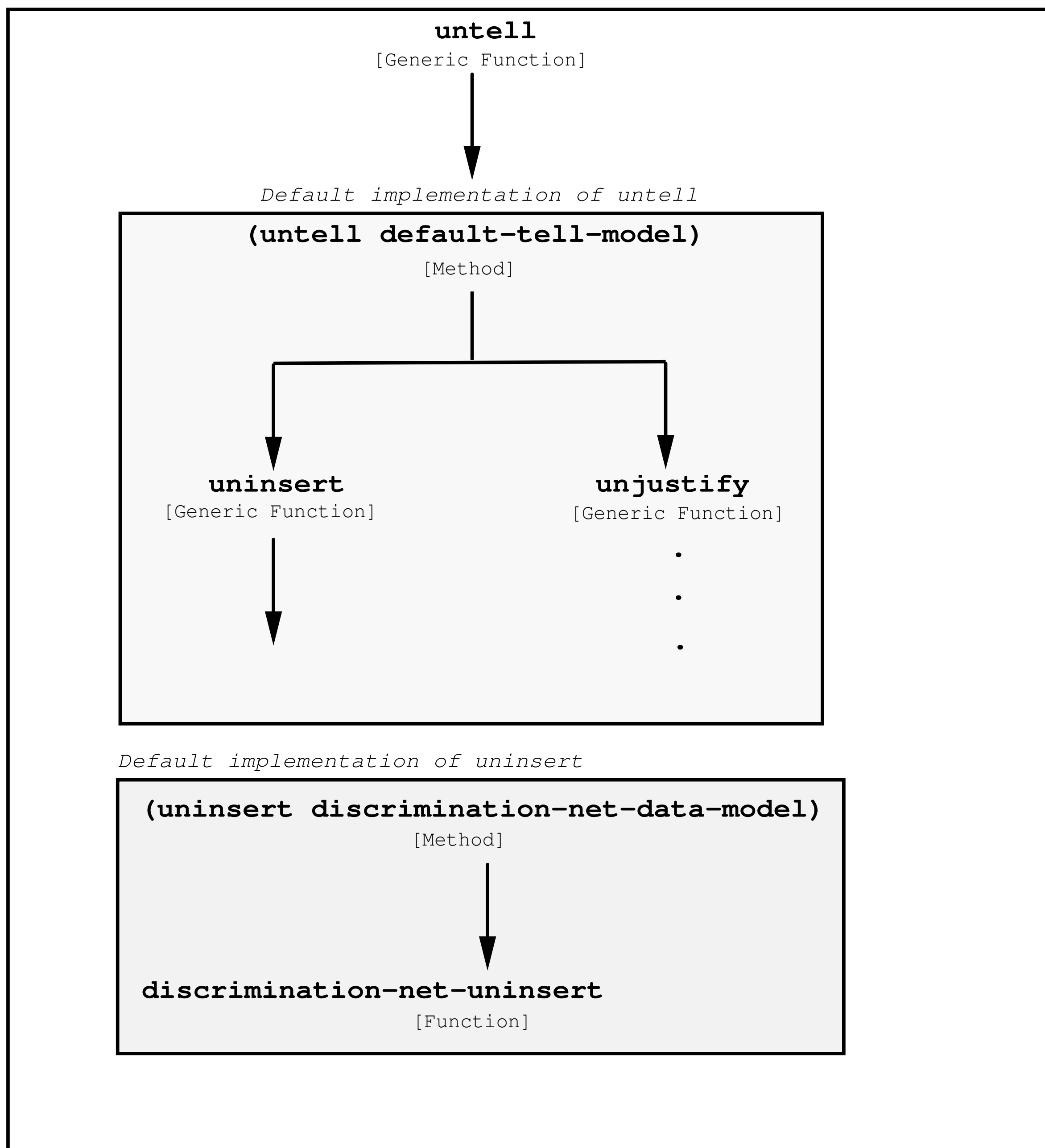


Figure 7. The **untell** protocol and its default implementation

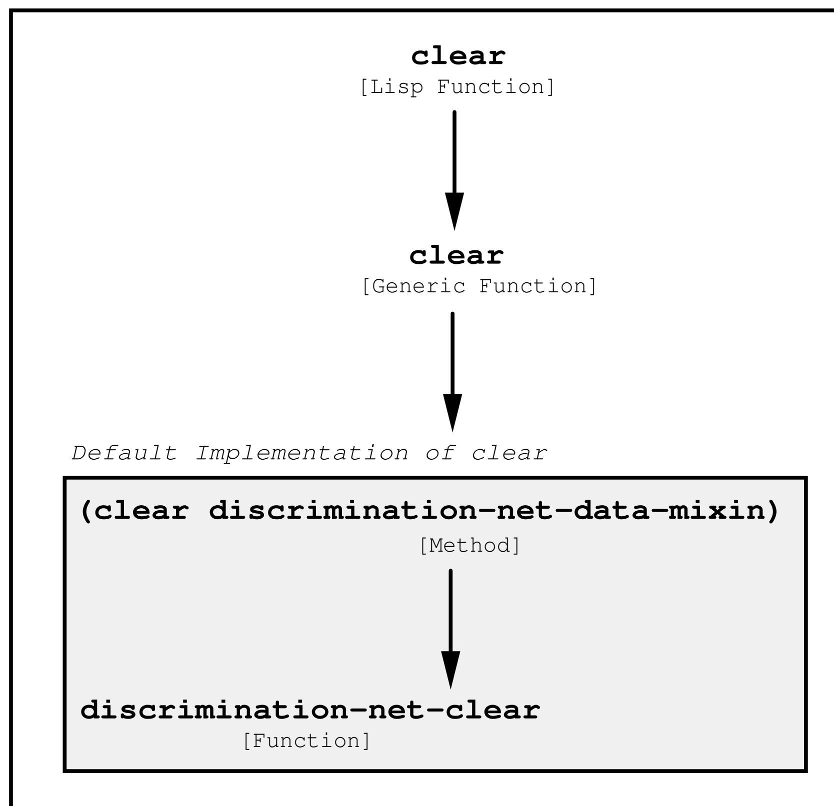


Figure 8. The **clear** protocol and its default implementation

2.4. Joshua's Default Database: the Discrimination Net

Joshua uses a data structure called a discrimination net for data storage and retrieval. This is a standard, domain independent data structure; it is organized so that in general the time needed to look up an item is independent of the number of items contained in the database. (In some specific problems you can do better than a discrimination net. See the section "Customizing the Data Index", page 81.)

The default discrimination net is written to support the basic model, **joshua:discrimination-net-data-mixin**.

Note: A good introduction to the practical matters of discrimination networks can be found in Eugene Charniak, Christopher K. Riesbeck, and Drew V. McDermott, *Artificial Intelligence Programming*, second edition (New Jersey: Lawrence Erlbaum Associates, 1987), chs. 8 and 11. In chapter 11 the authors discuss eight design decisions involved in the creation of a discrimination net. Here is the list of these, and the Joshua designers' choice in each case:

1. Are variables allowed in the data patterns? Yes
2. Are variables allowed in the query patterns? Yes
3. Does one keep track of variable bindings during fetching? No.
4. Should one return a list or a stream of possibilities? Pass closure down into dn fetcher.
5. Should one use CAR or CAR-CDR indexing? CAR. (Except for tail variables.)
6. Should one uniquify subexpressions? Yes.
7. Should one completely discriminate the data? Yes.
8. Should one use multiple indexing? No.

Please refer to the aforementioned book for further details.

2.4.1. Organization of the Default Discrimination Net

The organization of a storage structure such as a discrimination net has to do with the way in which the structure differentiates (*discriminates*) the objects that it stores. The discrimination net is organized to limit the search by eliminating invalid search targets. This is called *associative lookup*; it answers queries like, "find everything in the database that looks like this."

To see how the discrimination net stores predications, display a graph of the discrimination net with the form:

```
(graph-discrimination-net ji:*data-discrimination-net*)
```

The graphic representation of the database can also be a useful debugging aid if you are debugging an advanced model that calls the discrimination net, or if you suspect a performance bottleneck in the discrimination net. (You might, for example, look for a node with an unnecessarily large number of inferiors.)

The argument **ji:*data-discrimination-net*** contains the root node of the discrimination net to be graphed. The default root contains the token **ji::*begin-predication***; this merely stands for an object that begins predications in the discrimination net.

Figure 9, page 18 shows a sample graph display of a database containing predication objects with various arguments (lists, logic variables, nested predications, number, string, constant), to show how they are stored.

```
[hobby al (eating sleeping)]             ;list argument
[hobby jane (sailing skiing hiking)]     ;list argument
[foo ?x ?x]                              ;logic variable arguments (repeated)
[foo ?x ?y]                              ;logic variable arguments
[foo 1 [doodle 2]]                       ;nested predication argument
[foo 1 [doodle ?x]]                      ;nested predication with logic variable
[foo bar ?x]                             ;logic variable argument
[foo bar 2]                              ;numeric argument
[alcohol-content vodka "100%"]           ;string argument
[has-eye-color jane brown]
[has-eye-color fred green]
```

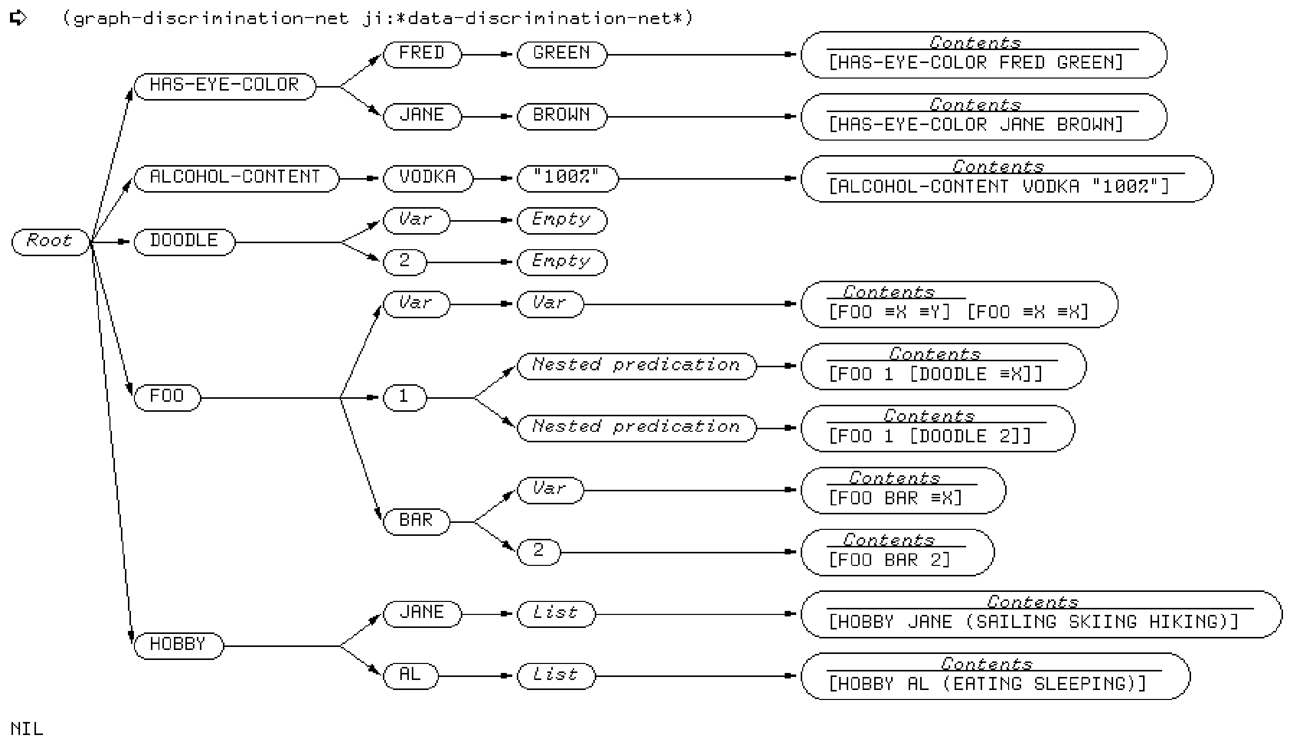


Figure 9. Sample Discrimination Net Display

The net looks like a tree seen horizontally, with the root node at the leftmost side. The immediate descendants of the root are predicates. The leaf nodes list the predication(s) that are stored in that node.

When more than one predication is built from the same predicate (as is the case with `[has-eye-color ...]` and `[foo ...]` in the figure), the tree branches, with separate branches discriminating the arguments for each predication.

Thus, if we are looking for a predication built on `[foo ...]`, the retrieval function **joshua:discrimination-net-fetch** (called by the **joshua:fetch** method of

joshua:discrimination-net-data-mixin) can ignore all predication branches other than those starting from [foo ...]. The search area is further narrowed down while searching a predicate tree: if you are looking for predication pattern [foo bar ...], the lookup function can ignore the other branches of [foo ...], and travel only through the arguments branching off from the bar node.

Figure 9 shows the individual storage of constant arguments, string arguments, numeric arguments, and nested predications. When the discrimination net encounters a nested predication like [foo 1 [doodle 2]] in the graph example, it rediscriminates the nested predication from the root, and uses the resulting (unique) leaf node as a token for the nested predication. The nested predication also appears in the containing predication; here it is stored in a node labelled *Nested predication*.

Discrimination of arguments helps to limit the number of items that are retrieved. So, for example, if you want to look up all persons with green eyes, **joshua:fetch** traverses the branches for both [has-eye-color Fred ...] and [has-eye-color Jane ...], but rejects the latter because the color arguments don't match. Here are some examples using constant and string arguments.

```
; discrimination net filters non-matching constant and string arguments
  (fetch [has-eye-color ?person green] #'print)
[HAS-EYE-COLOR FRED GREEN]
NIL

  (fetch [has-eye-color jane ?color] #'print)
[HAS-EYE-COLOR JANE BROWN]
NIL

  (fetch [has-eye-color ?who ?color] #'print)
[HAS-EYE-COLOR FRED GREEN]
[HAS-EYE-COLOR JANE BROWN]
NIL

  (fetch [alcohol-content ?x "100%"] #'print)
[ALCOHOL-CONTENT VODKA "100%"]
NIL

  (fetch [alcohol-content vodka ?x] #'print)
[ALCOHOL-CONTENT VODKA "100%"]
NIL
```

Two types of arguments, namely, lists and logic variables, are not stored individually. This is reflected in the way **joshua:fetch** deals with queries containing such arguments.

All lists are equivalent to the discrimination net. They are stored in a node whose token is **ji::*embedded-list***. The grapher displays this as *List*. That is, all lists look pretty much the same to the discrimination net. See the branches for predicate hobby in figure 9.

Since lists are not discriminated at the database level, and the lookup function is not responsible for unification, **joshua:fetch** gets all possibilities when dealing with list arguments. In the example below, even though we specify to **joshua:fetch** the exact pattern to find, we get everything that starts with the target predicate, including answers that don't necessarily unify with the query.

```
; discrimination net does not discriminate lists
; fetch gets all possible answers
(fetch [hobby ?x (eating sleeping)] #'print)
[HOBBY JANE (SAILING SKIING HIKING)]
[HOBBY AL (EATING SLEEPING)]
NIL
```

Since no unification is done at the database level, logic variable arguments are not discriminated, but rather stored in a node whose token is **ji::*variable***. The grapher displays this as *Var*. That is, all variables look pretty much alike to the discrimination net. So, for example, although the logic variables in `[foo ?x ?x]` and `[foo ?x ?y]` are distinct to the unifier, both predications are stored identically, as we see in figure 9, and both appear in the same rightmost node.

As it does with lists, the lookup function gets all possibilities when dealing with logic variables either in the lookup pattern or in the database.

```
; discrimination net does not discriminate logic variables
; fetch gets all possible answers

  (fetch [foo ?x ?x] #'print)
    ;; this finds six answers, even though only three will pass
    ;; the unification test in ask-data
[FOO 1 [DOODLE ?X]]
[FOO 1 [DOODLE 2]]
[FOO BAR 2]
[FOO BAR ?X]
[FOO ?X ?Y]
[FOO ?X ?X]
NIL
```

```
  (fetch [foo 1 ?x] #'print)
[FOO 1 [DOODLE ?X]]
[FOO 1 [DOODLE 2]]
[FOO ?X ?Y]
[FOO ?X ?X]
NIL

  (fetch [foo bar 2] #'print)
[FOO BAR 2]
[FOO BAR ?X]
[FOO ?X ?Y]
[FOO ?X ?X]
NIL
```

3. The Joshua Rule Facilities

The basics of rule syntax and operation were presented in the section "Rules and Inference". Figure 10, page 23 summarizes the main points of this earlier discussion here, for quick reference.

Rule Type	Triggered by	Trigger Part	Predications In Trigger	Rule Fires When	Action Part	Rule Success (all if-parts satisfied)
Forward	**TELL**	**if-part**	**Compound**	**Triggers Satisfied**	**then-part**	**Rule executes action part**
Backward	**ASK**	**then-part**	**Single**	**Trigger Asked**	**if-part**	**ASK calls continuation**

Figure 10. Summary of Joshua Rule Operation

This chapter amplifies some of the basics, and discusses the *rule compiler* and *rule indexing*. The rule compiler translates your rules into Lisp. Rule indexing is the way the system adds, deletes, and finds rules.

Given an initial set of facts, rules allow us to infer or deduce additional new facts, that are consequences or conclusions. Since rules define how the system reasons about its knowledge, they are one means of controlling the acquisition and extension of current knowledge.

Joshua programs can use either *forward chaining* or *backward chaining* rules, or both. Forward chaining is triggered by a **joshua:tell** statement and its action(s) can result in more **joshua:tell** statements, adding newly deduced facts to the database. Backward chaining is triggered by an **joshua:ask** statement, and its action(s) involve the **joshua:ask** mechanism, that is, it works from existing data rather than inferring new data.

Seen declaratively, a forward or backward rule is simply a special kind of fact, stating a logical truth. In other words, there is nothing special in terms of logic to distinguish a forward from a backward rule, since both are derived from modus ponens. Seen mechanistically, there is a distinction between forward and backward rules, in terms of the processing directives each gives the system.

There are many ways of doing inference. For most engineering applications, however, forward and backward chaining are computationally efficient problem solvers that balance the power of mathematical logic with the efficiency of the computing mechanism. Both of these rules are *sound*, but they are *incomplete*. Soundness means that using these rules of inference always produces logically (mathematically) correct conclusions. Incompleteness means that, in some cases, some correct conclusions will not be found.

Soundness is obviously a desirable characteristic. Incompleteness is undesirable, but complete rules of inference are computationally less efficient; also for most practical applications incompleteness doesn't seem to be a problem. (One application area which does require completeness is mathematical theorem proving.)

In general, then, the Joshua programmer need not worry about these formal properties of the Joshua inferencing mechanism. But looking at your program in terms of logic can give you some guidance in debugging it. In brief, we can separate bugs into four categories:

1. Drawing incorrect conclusions.

2. Not drawing correct conclusions.

3. Non-termination ("infinite loops").

4. Errors detected when rules call Lisp functions incorrectly.

Soundness tells us that if our system is drawing incorrect conclusions, there must be an incorrect piece of data or an incorrect rule.

When the system is failing to draw some correct conclusion, it may be due to incorrect data or rules, or it may be due to the incompleteness of the rule of inference. In that case, it is necessary to change the program in ways which leave it logically the same, but change the structure enough to allow the inference mechanisms to cope with it. This change can either be adding rules or data which are logically redundant, or it can be restructuring the rules or data.

3.1. Advanced Features of Joshua Rules

This section summarizes the full syntax of both forward and backward chaining rules.

Both forward and backward rules allow various keywords to be attached to the patterns of the If-part of the rule. Both Forward and Backward rules allow the Keyword :support followed by a logic-variable:

```
(defrule foobar (:forward)
  If [and [foo ?x ?y] :support ?f1
          [bar ?y ?z] :support ?f2]
  Then (format t "~&I won with F1 = ~s and F2 = ~s" ?f1 ?f2))

(defrule foobar (:backward)
  If [and [bar ?x ?y] :support ?f1
          [bar ?y ?z] :support ?f2]
  Then [foo ?x ?z])
```

This indicates that the logic-variable should be bound to the "support" for this pattern. In the case of a forward rule, the support is simply the fact which matched the corresponding pattern. Thus

```
(tell [and [foo 1 2] [bar 2 3]])
```

will cause the first rule above to print:

```
I won with F1 = [FOO 1 2] and F2 = [BAR 2 3]
```

Backward rules turn their If-part into a series of nested **joshua:ask**'s. When the first **joshua:ask** finds a match, it calls a continuation which performs the next **joshua:ask**. The argument to this continuation is a "backward-support" structure, see the section "Continuation Argument", page 125.

The support keyword in a backward rule binds the logic-variable to the backward support corresponding to its query.

Thus with the following rule and data:

```
(defrule foobar (:backward)
  If [and [bar ?x ?y] :support ?f1
          [bar ?y ?z] :support ?f2
          (progn (format t "~&I won with F1 = ~s and F2 = ~s" ?f1 ?f2)
                 (succeed))
          ]
  Then [foo ?x ?z])

(tell [and [bar 1 2] [bar 2 3]])
```

The query:

```
(ask [foo 1 3] #'print-query)
```

will cause the following output:

```
I won with F1 = ([BAR 1 2] 1 [BAR 1 2]) and F2 = ([BAR 2 3] 1 [BAR 2 3])
[FOO 1 3]
```

This backward support may be used to provide a justification (for a TMS) when a backward rule caches the results of its work, as follows:

```
(defrule foobar (:backward)
  If [and [bar ?x ?y] :support ?f1
          [bar ?y ?z] :support ?f2
          (progn
            (tell [foo ?x ?z]
                  :justification `(foobar
                                   (,(ask-database-predication ?f1)
                                    ,(ask-database-predication ?f2))))
            (succeed))
          ]
  Then [foo ?x ?z])
```

Backward rules also support two other keywords **:do-backward-rules** and **:do-questions**. These can be used to control the behavior of the **joshua:ask** corresponding to a backward action. If the **:do-backward-rules** keyword is present then the value following it should evaluate to either **joshua::t** or **joshua::nil**; if it is **joshua::nil**, then this query will not attempt to use rules to satisfy the query, otherwise rules will be used. Similarly, the **:do-questions** questions controls whether backward questions will be invoked to query the user. The default value is that backward rules are used and that questions will be attempted if the query which caused this rule to be invoked allowed questions to be used.

3.2. The Joshua Rule Compiler

The *rule compiler* is the part of Joshua that translates the rules you write into Lisp. This section describes, in general terms, how the rule compiler operates. Knowledge of how the rule compiler operates is important if you would like to extend the rule compiler later on by defining your own methods for the rule compiler's generic functions.

The rule compiler uses several generic functions to generate data structures that drive the rest of rule compilation. These five generic functions are as follows:

joshua:expand-forward-rule-trigger
joshua:expand-backward-rule-action
joshua:write-backward-rule-matcher
joshua:write-forward-rule-semi-matcher
joshua:write-forward-rule-full-matcher

The functions handle forward chaining and backward chaining for both triggers and actions. Recall that the trigger of a forward chaining rule is the *if-part*, and the action is the *then-part*. For backward-chaining rules, the trigger is the *then-part*, and the action is the *if-part*. See the section "Rules and Inference" in *User's Guide to Basic Joshua*.

Any user-written **joshua:defrule** expression expands into two things:

- *Trigger code* that decides when to execute the rule

- A function, written by the rule compiler, that becomes the *rule body*.

3.2.1. The Forward Rule Compiler

In this section we examine what happens when the rule compiler encounters a forward chaining rule of the form:

```
(defrule <name> (:forward <...>)
        if <trigger>
        then <action>)
```

(The code within broken brackets is a schematic representation of the actual code you supply.)

3.2.1.1. Compiling the Action Part of a Forward Rule

The action part of a forward rule is not generic, it is always handled the same way:

1. If the action is a Lisp form, insert it into the rule body.

2. If the action is a predication, (except an **joshua::and**) insert

   ```
   [tell predication :justification
                     <a justification that depends on the triggers>]
   ```

   If the action is an **joshua::and** predication, recurse over its arguments as in (1) and (2).

3.2.1.2. Forward Rule Triggers: the Rete Network

A forward chaining rule should fire as soon as its trigger is matched by predications in the database. However, the trigger part of a rule typically consists of several patterns linked by the connective **and**. For such a rule to fire, two conditions must be met:

1. Each pattern must match some predication in the virtual database.

2. The matches must be consistent. Any logic variable which is bound by matching one of the patterns must be bound to the same value by all of the patterns.

Since each pattern of the rule may match several predications in the database (and these matches lead to different bindings of the logic variables), finding consistent sets of matches inherently involves a form of search.

The firing of a forward chaining rule should not depend on the order in which the predications which match the rule's patterns are asserted. A rule should fire as soon as a consistent set of matches is available. In effect, each time a predication changes truth-value, a search is conducted for such consistent matches. The task of the rule compiler is to build a data-driven structure which can conduct this search in an efficient manner whenever the truth-value of a predication in the virtual database changes. This stucture is called a *Rete Network*; the triggers of a forward rule are *nodes* in this network, specifically *match nodes*.

(Rete networks were originally used in *Production System* languages such as OPS-5. For more information on Rete networks, see C. Forgy 1982. "Rete: A Fast Algorithm for the Many Pattern/Many Object Pattern Match Problem." *Artificial Intelligence* 19: pp. 17-37.)

Here's an example rule with a multiple trigger -- a set of predications linked by **and**:

```
(defrule example (:forward)
  if [and [foo ?x]
          [bar ?x ?y]
          [bar ?y ?z]]
  then <some-action>))
```

This rule should fire twice if the following predications are in the database:

```
[foo 1]
[bar 1 2]
[bar 2 3]
[foo 2]
[bar 3 4]
```

One firing should have the follow binding of logic variables:

```
?x → 1, ?y → 2, ?z → 3
```

The second firing should have the following set of bindings:

```
?x → 2, ?y → 3, ?z → 4
```

The rete network accomplishes this triggering relatively efficiently. This rule's Rete network conceptually looks like the drawing in figure 11. Note how the variable bindings, denoted in braces, flow down, until the rule is fully triggered.
This Rete network consists of two types of nodes: *match nodes* and *merge nodes*.

Match nodes contain *match procedures* generated by the rule compiler; each match procedure corresponds to a particular pattern of the rule. When a predication changes truth-value (and assumes a definite truth-value for the first time), the match node is located in the rule index. See the section "The Joshua Rule Indexing Protocol", page 36. The match procedure is then invoked with the predication as argument; its job is to determine if the predication can be unified with its corresponding pattern. If the unification succeeds, the match procedure returns a *binding environment* which maps each logic-value to the value assigned to it by the unification. This environment is remembered in the match node.

Match nodes also contain pointers to subordinate merge nodes. The match node passes the binding environments stored in it down to each merge node subordinate to it.

Merge nodes receive binding environments from their left and right inputs and, if possible, produce a consistent extension of those environments as output. When a new environment is sent to a merge node from a left parent, the merge node checks this environment against every environment stored in its right parent. Similarly, when an environment arrives from the right parent, the merge node checks it against every environment stored in the left parent.

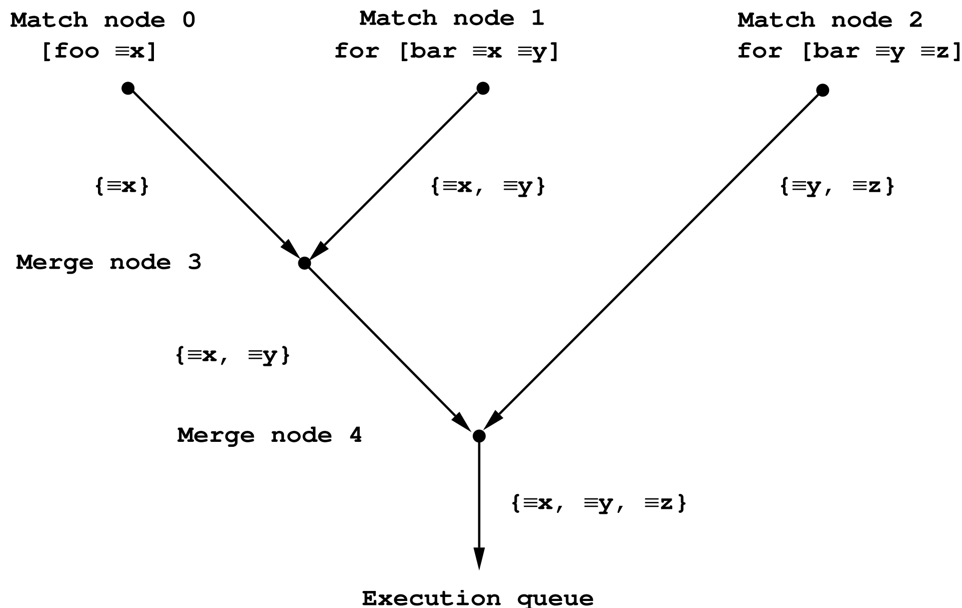


Figure 11. Sample Rete Network

The check for consistency between pairs of environments is performed by a *merge procedure* generated by the rule compiler and stored in the merge node. Each merge procedure corresponds to a set of patterns. In the rule shown in figure 11 the merge nodes correspond to the sets:

```
[foo ?x] [bar ?x ?y] --> Merge Node 3

[foo ?x] [bar ?x ?y] [bar ?y ?z] --> Merge Node 4
```

In effect, each merge node in this Rete network adds in the bindings resulting from the matching of one additional pattern. This is normally the case; however, the use of a nested group of patterns connected by **and** can lead to other merge patterns.

Each Merge node contain pointers to subordinate merge nodes. When the merge node successfully unifies a pair of binding environments, it creates an extended environment which it stores in the node; it also sends the extended environment to each merge node subordinate to it. A merge node which corresponds to the full set of patterns in the trigger part of the rule initiates execution of the rule.

To summarize, match nodes are the triggers produced by the rule compiler. They are invoked when a predication first assumes a definite truth-value (as the result of **joshua:tell** or **joshua:justify**, for example). A match node checks if the predication matches a particular pattern. If so, it produces an environment which is sent to subordinate merge nodes. These check if the environments produced by matching different patterns are consistent (in the sense that they produce compatible logic variable bindings). Merge nodes pass their environments along to other merge nodes until a terminal merge node for a rule is reached. At that point all

the patterns for the rule have been consistently matched and the rule may be executed. All these environments are remembered, so that rules can be partially triggered and not have to redo the partial trigger combination.

The command Graph Forward Rule Triggers lets you display the Rete network for the rule (s) specified. Figure12shows the graph for the rule EXAMPLE.

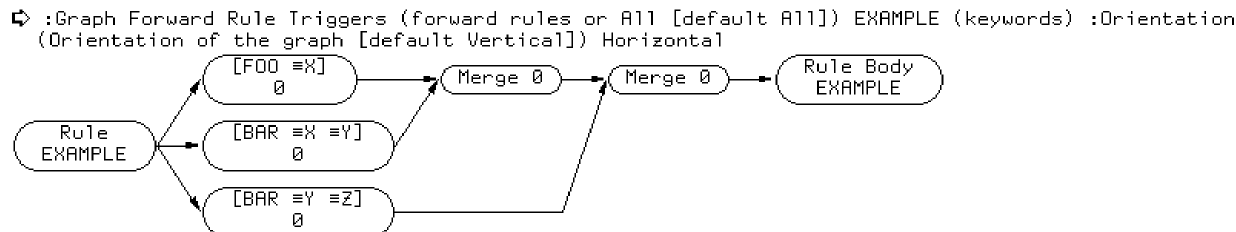


Figure 12. Sample Rete Network Display

Joshua's Rete networks actually include four kinds of nodes, *match* nodes, *merge* nodes, *procedural* nodes and *or* nodes. We have already seen match and merge nodes; we now turn to the other two types of rete network nodes.

Some rules include *procedural triggers*, i.e. Lisp code in the *if*-part of the rule. Procedural triggers can play either of two roles: *filters* and *generators*. A filter is a piece of Lisp code that returns **t** or **nil** but does not bind new logic variables. A generator is a piece of Lisp code that binds new logic variables (using **joshua:ask** or **joshua:unify**) and calls **joshua:succeed** (possibly many times). Calling **joshua:succeed** "endorses" the current variable bindings, thus allowing the rule body to execute with those bindings.

The Rete networks for these rule have *procedural nodes* corresponding to this code.

Here's a simple example of a procedural trigger, acting as a filter on the previous bindings:

```
(defrule filter-example (:forward)
  if [and [foo ?x]
          (> ?x 5)
          [bar ?x ?y]]
  then <some-action>))
```

Figure 13 shows the graph for the rule filter-example.

This filter ensures that the logic variable ?x is bound to a value larger than 5.

Within the filter you can refer to the values bound to the logic variables in previous patterns.

Generators can side-effect the current variable bindings and they can **joshua:succeed** several times. Here is an example of a generator:

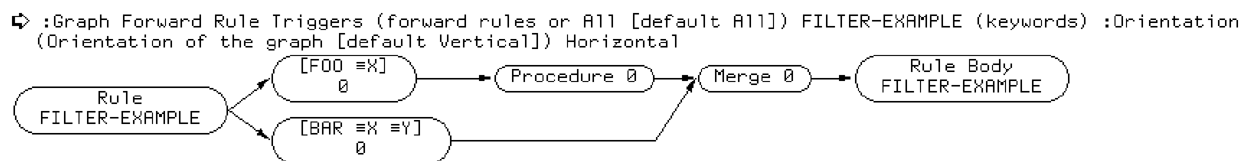


Figure 13. Sample Rete Network Display with Filter Nodes

```
(defrule eating-test (:forward)
   If [and [good-to-eat ?x]
           [hungry ?y]
           (loop for eating-mode in '(orally intravenously) do
                 (with-unification
                   (unify ?z eating-mode)
                   (succeed)))
           [can-eat ?y ?z]]
   Then ...)
```

Here the triggers check to see if ?x is something good to eat and ?y is hungry. For every "eating mode" that ?y can use, the rule unifies ?z to that mode and calls **joshua:succeed**.

When a forward chaining rule uses **or** to link together trigger patterns, the rule compiler builds an *or node* in the Rete network. For example, the rule above could have also been written as follows:

```
(defrule eating-test (:forward)
   If [and [good-to-eat ?x]
           [hungry ?y]
           [or [eating-mode ?y orally]
               [eating-mode ?y intravenously]]
           [can-eat ?y ?z]]
   Then ...)
```

Then we would get a Rete network with an *or node* which joins the two match nodes for the [eating-mode ?y orally] and the [eating-mode ?y intravenously] patterns, as shown in figure 14.

The forward rule syntax allows arbitary nesting of groups of patterns linked by **and** and **or**. Semantically there is no reason to nest one **and** group within another; however, there is a procedural difference.

Each nested **and** group forms its own sub rete network (i.e its own merge group) which is then merged with the patterns from the enclosing group. Consider the following rule with a nested **and** group:

```
(defrule nested-and (:forward)
  if [and [foo ?x]
          [and [bar ?x ?y]
               [bar ?y ?z]]]
  then < ... >)
```

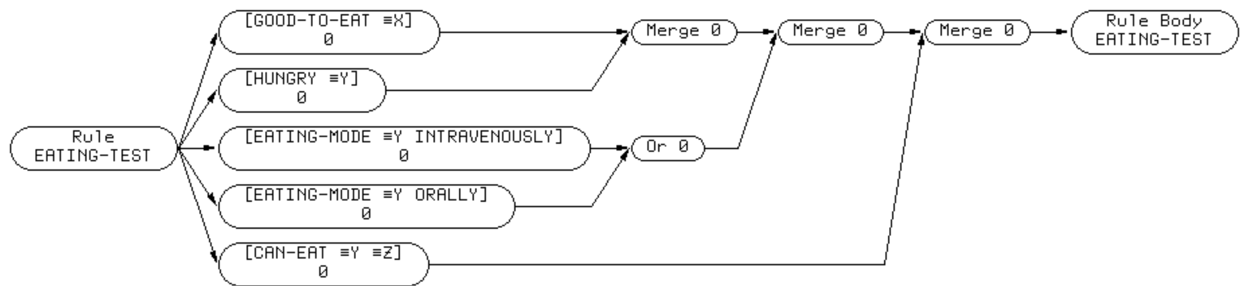


Figure 14. Sample Rete Network Display with **or** Node

The rete network for this rule is:

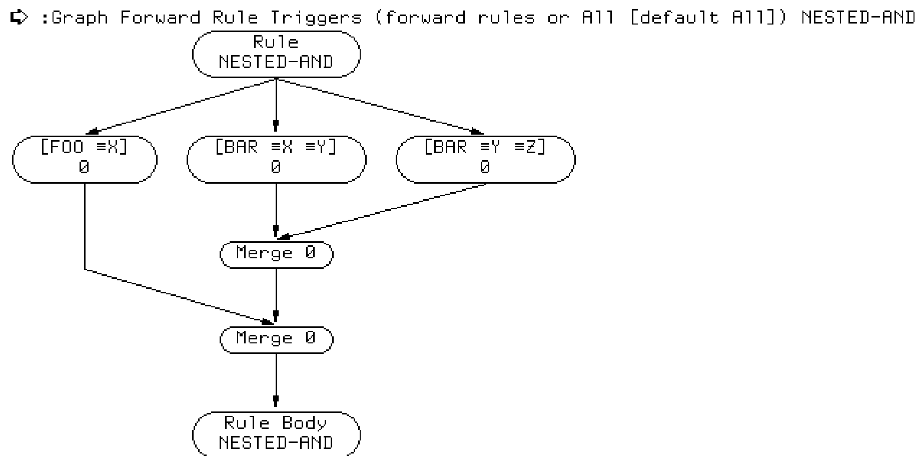


Figure 15. Rete Network For Rule with Nested Ands

Notice that the two BAR patterns merge first and then the result of this is merged with the FOO patterns. Suppose that it is unlikely that the results of matching the two BAR patterns will successfully merge but that it is very likely that the result of matching the FOO pattern will almost certainly merge successfully with the result of matching the first BAR pattern. The rete network shown will improve efficiency by not generating a merged environment resulting from matching the first two patterns when it is likely that this intermediate result will not successfully merge with the result of matching the last pattern.

In most cases, the code generated by the rule compiler can be made significantly more compact using a few simple techniques,see the section "Optimizing Forward Rule Compilation for Semi Unification".

The forward rule compiler may be customized by use of the Joshua protocol functions: **joshua:expand-forward-rule-trigger**, **joshua:write-forward-rule-full-matcher**, **joshua:write-forward-rule-semi-matcher**, and **joshua:positions-forward-rule-matcher-can-skip**.

The **joshua:expand-forward-rule-trigger** protocol function allows you to control how trigger patterns are compiled by first allowing you to expand the trigger into a set of expressions which are understood by the rule compiler.

The **joshua:write-forward-rule-full-matcher** and **joshua:write-forward-rule-semi-matcher** protocol functions allow you to control the generation of the pattern matching code corresponding to each forward rule trigger.

The **joshua:positions-forward-rule-matcher-can-skip** protocol function is called by **joshua:write-forward-rule-semi-matcher**; it allows you to provide advice to the match generator about what parts of a pattern have already been checked by the rule indexer.

See the section "Customizing the Rule Index", page 88.

3.2.2. The Backward Rule Compiler

This section describes the rule compiler's operation on backward chaining rules. Here, in schematic form, is a backward chaining rule.

```
(defrule <name> (:backward <...>)
        if <action>
        then <trigger>)
```

As this indicates, the trigger of a backward chaining rule is the then-part of the rule. A backward trigger is a single predication. The matcher for the trigger of a backward chaining rule is simple:

```
match <trigger>
      |
      v
invoke rule body
```

For the action part of a backward chaining rule, the rule compiler creates a function that is called when the rule is invoked.

```
(defun <name> (args ... <continuation> ...)
        ... BODY ...
        (funcall <continuation>)
```

Calling the <name> is referred to as *firing* the rule, and calling the <continuation> is referred to as *succeeding*. The BODY decides when to call the continuation, this being the main task of the action part of a backward chaining rule.

Three types of actions are possible.

1. The action is a single predication (excluding **joshua::and**), and the rule compiler generates an **joshua:ask** for that predication:

   ```
   (ask <predication> <continuation>)
   ```

2. The action includes multiple predications joined by an **joshua::and**

```
[and p1 p2 ... pn]
```

and the rule compiler generates a set of nested **joshua:ask** functions:

```
(ask p1
     #'(lambda (p1-support)
         (ask p2
              ...
                (ask pn ...))))
```

Note that the last continuation (the innermost **joshua:ask**) is the continuation argument to the rule.

3. The action part of a backward chaining rule is a Lisp form. In this case, the form is called and if it returns non-**nil**, the firing of the rule goes on. Users can force rule firing to continue by having the form call **joshua:succeed** explicitly. Here is an example of this kind of action.

```
(defvar *known-foods* '(chinese-food))
(defrule good-to-eat-rule (:backward)
 if (typecase ?x (sys:unbound-logic-variable)
       ;; if ?x is unbound succeed once for every element of *known-foods*
       (loop for food in *known-foods*
              doing (unify ?x food)
              (succeed))
       ;;the case where ?x is bound is omitted.
       ... )
 then [good-to-eat ?x])
```

In this way, the procedural Lisp code in the *if*-part can act either as a filter or a generator. This treatment of Lisp code is the same as for forward rules. See the section "The Forward Rule Compiler", page 27. The person that calls this rule is interested in obtaining a list of everything that is good to eat or in finding out if a specific food is good to eat.

The behavior of the backward rule compiler can be customized by use of 2 Joshua protocol functions: **joshua:expand-backward-rule-action** and **joshua:write-backward-rule-matcher**.

The **joshua:expand-backward-rule-action** protocol function allows you to control how the action part of a backward rule (i.e. the If-part) patterns are compiled by first allowing you to expand the actions into a set of expressions which are understood by the rule compiler.

The **joshua:write-backward-rule-matcher** protocol function allows you to control the generation of the pattern matching code corresponding to the backward rule's trigger (i.e. its **then** part).

3.3. Ordering Rule Execution

When you define a rule, the keyword **:importance** lets you specify the order of rule execution. This keyword takes a value that can be any numeric argument, a symbol, or a form. The larger the number, the higher the priority. High priority rules run first.

Some expense is associated with using **:importance**. In forward chaining rules ordering causes a "best-first" search of rules according to the value associated with **:importance**. Backward chaining only orders the local "best-first" search of related rules.

Using **:importance** is convenient, but reduces efficiency. The system is most efficient when only one rule at a time is applicable. A situation where more than one rule is applicable usually indicates that insufficient knowledge is built into the rules. For example, a picture-taking program might have three separate rules responding to a request for a picture: one rule focuses the camera, another reads the light meter, and another sets the time and aperture. If you now tell the system to take a picture, it will not know which of the three rules to execute first. Although the **:importance** feature could be used to order the execution of these rules, it would be clearer and more robust to make the rules more explicit about their preconditions, thereby restricting their applicability (your focusing rule might only trigger if the light meter shows acceptable readings, and so on).

The **:importance** feature can be quite useful to control performance tradeoffs, such as trying a cheap algorithm first, in preference to more expensive algorithms. (Readers familiar with the *production system* model used in OPS-5 will recognize that the **:importance** feature serves a similar role for forward rules to the *conflict resolution strategy* of a production system).

Sometimes it is useful to be able to suspend forward rule triggering until the execution of a block of code has completed. The code might contain a number of **joshua:tell**'s and **joshua:untell**'s intermixed in such a way that the changes to the database are not coherent until the entire block of code has finished executing. Joshua provides a special form which offers this form of control. See the macro **joshua:with-atomic-action**, page 254.

3.4. Controlling Rule Invocation

Typically during a **joshua:tell** or an **joshua:ask**, the database is searched first (**joshua:insert**, or **joshua:fetch**, respectively), after which the appropriate trigger-mapping function (forward, backward, or backward question) is executed, to find and run relevant rules or questions. Several facilities are available to let you modify this sequence.

When you set the **joshua:ask** keyword **:do-backward-rules** to **nil**, backward rule invocation is inhibited, and the system does a database lookup only.

Example:

```
(define-predicate age (person age))
(define-predicate attained-majority (person))

(defrule old-enough (:backward)
  if [and [age ?person ?age]
          (> ?age 21)]
  then [attained-majority ?person])

(tell [age Fred 21])

(ask [attained-majority Fred] #'print-query :do-backward-rules nil)
NIL          ; information is not in the database

(ask [attained-majority Fred] #'print-query)
[ATTAINED-MAJORITY FRED]    ; backward rule is invoked
NIL
```

Six built-in flavors are also available for predicates used in **joshua:ask** goals. These flavors do subsets of what **joshua:ask** normally does, by leaving out one or more of the steps **joshua:ask-data**, **joshua:ask-rules**, or **joshua:ask-questions**. Thus the models save a certain amount of overhead when their predicates are used as goals to **joshua:ask**. The steps which *are* done are indicated by the names:

- **joshua:ask-data-only-mixin**
- **joshua:ask-rules-only-mixin**
- **joshua:ask-questions-only-mixin**
- **joshua:ask-data-and-rules-only-mixin**
- **joshua:ask-data-and-questions-only-mixin**
- **joshua:ask-rules-and-questions-only-mixin**

3.5. The Joshua Rule Indexing Protocol

Joshua manages rules by manipulating rule triggers. There are four trigger operations, namely:

- Adding triggers
- Deleting triggers
- Locating triggers
- Iterating over triggers

Rule indexing refers to the protocol steps that determine how these rule operations are performed.

Often systems spend most of their time looking for applicable rules, as opposed to executing them. If this is the case, customizing the trigger index can help. This is the process of changing the way the system stores, removes, looks up, and iterates over triggers. If you provide a consistent alternative implementation of these actions, you have changed the way your program looks for rules. This is discussed in detail elsewhere:See the section "Customizing the Rule Index", page 88.

This chapter covers the protocol and implementation details that you need to know about before you attempt to customize the rule index. If you are using the default rule index, you may not find these topics of immediate interest.

The rule indexing protocol uses separate functions for forward and backward rule indexing operations. Here is the list of functions.

joshua:add-forward-rule-trigger Add a forward rule trigger

joshua:add-backward-rule-trigger Add a backward rule trigger

joshua:delete-forward-rule-trigger Remove a forward rule trigger

joshua:delete-backward-rule-trigger Remove a backward rule trigger

joshua:locate-forward-rule-trigger Find a forward trigger data index, and calls a continuation on it.

joshua:locate-backward-rule-trigger Find a backward trigger data index, and calls a continuation on it.

joshua:map-over-forward-rule-triggers Call a continuation on all forward triggers that might unify with a given predication

joshua:map-over-backward-rule-triggers Call a continuation on all backward triggers that might unify with a given predication

Figure 16 shows how the rule-indexing functions relate to each other.

Like the data indexing functions, the trigger indexing functions work together in the sense that they must share a knowledge of the trigger storage location. The adding functions must install a trigger in a place such that the mapping and deleting functions can find it. The deleting functions must delete triggers from places where the mapping functions look for them.

The trigger object that is processed by the rule-indexing protocol is created by the rule compiler. Exactly what this object is depends on what kind of rule is involved. For a forward rule, the triggers are Rete match nodes. See the section "Forward Rule Triggers: the Rete Network", page 27. Invoking a trigger means a call to some Rete network code to start the match process. For backward rules the

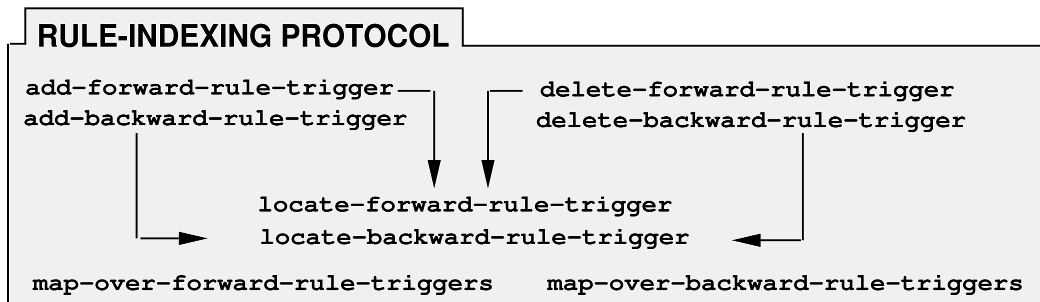


Figure 16. Rule Indexing Protocol

(unique) trigger is also a match procedure, but without some of the complications of the Rete mechanism.

The contract of the rule indexing functions is very similar to that of the data indexing functions. (See the section "The Joshua Database Protocol", page 8.)

3.5.1. The Contract of the Trigger Adding Functions

The protocol functions **joshua:add-forward-rule-trigger** and **joshua:add-backward-rule-trigger** are analogous to the data-indexing function **joshua:insert**. When a new rule is compiled, the compiler uses the appropriate version of the trigger adding functions (forward or backward) to add a trigger object to the data structure that holds trigger objects.

When a new forward trigger is installed, the database must be searched for facts that might match the new trigger. **joshua:add-forward-rule-trigger** does this database lookup by calling the protocol function **joshua:prefetch-forward-rule-matches**; the default version of this protocol function simply calls **joshua:ask** to find the appropriate facts.

In the default implementation, the finding, building, and updating of trigger storage structures is the responsibility of the trigger locating functions, **joshua:locate-forward-rule-trigger**, and **joshua:locate-backward-rule-trigger**. See the section "The Contract of the Trigger Locating Functions", page 39.

Figure 17 shows the protocol for the trigger adding functions and their default implementation.
See the section "Forward Rule Triggers: the Rete Network", page 27.

3.5.2. The Contract of the Trigger Deleting Functions

The protocol functions, **joshua:delete-forward-rule-trigger** and **joshua:delete-backward-rule-trigger** are analogous to the data-indexing function **joshua:uninsert**. These trigger deleting functions are used by **joshua:undefrule** to remove a trigger object from a list of triggers. The default implementation stores trigger information in a discrimination net.

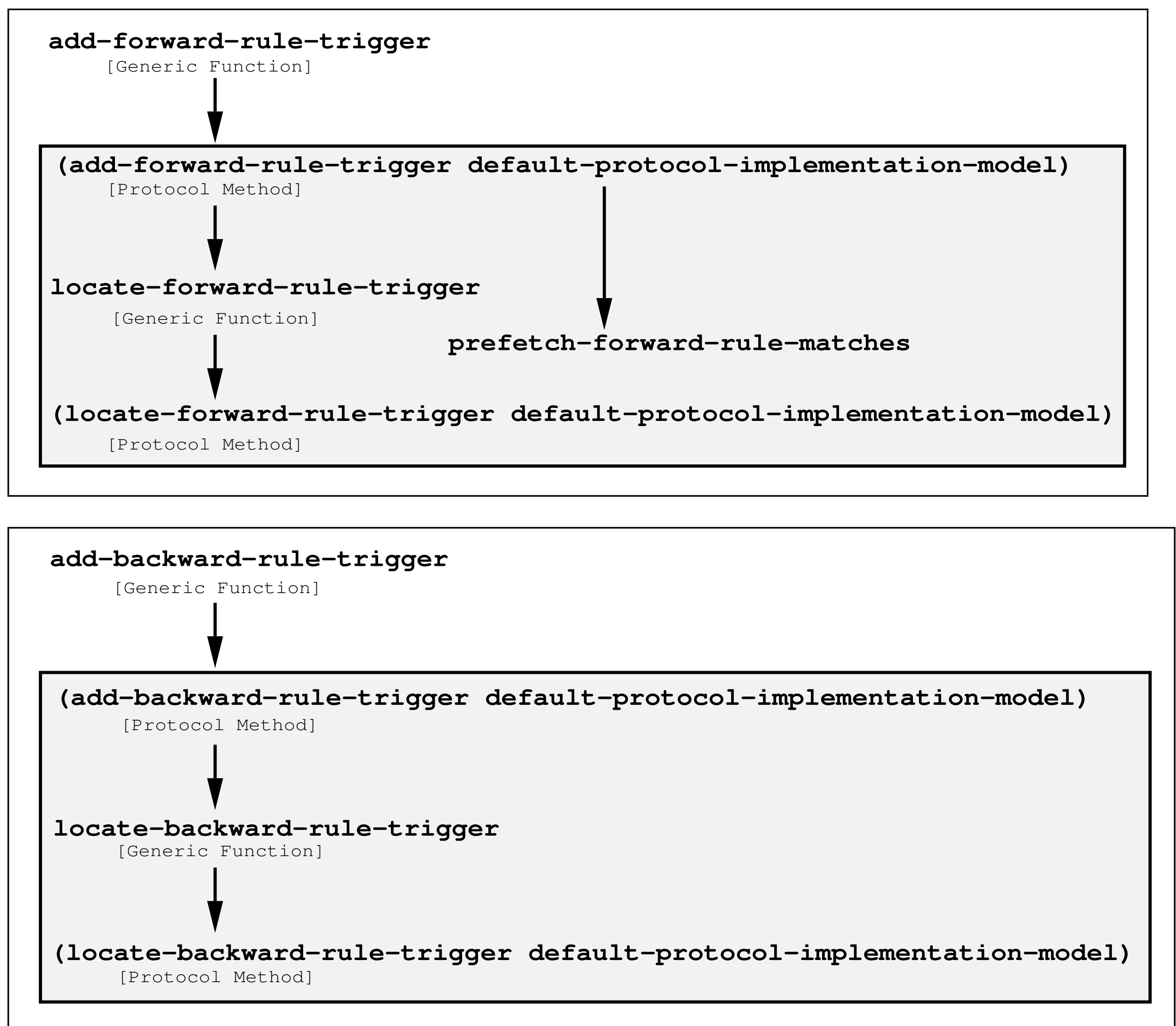


Figure 17. The Trigger-Adding Protocol and Default Implementation

The trigger deleting functions rely on the trigger locating functions to do the actual removal of the trigger and to update the trigger storage location. See the section "The Contract of the Trigger Locating Functions", page 39.

Figure 18 shows the protocol for the trigger deleting functions and their default implementation.

3.5.3. The Contract of the Trigger Locating Functions

The contract of the protocol functions **joshua:locate-forward-rule-trigger** **joshua:locate-backward-rule-trigger** and **joshua:locate-backward-question-trigger** is to *find, build, and update* forward and backward trigger storage struc-

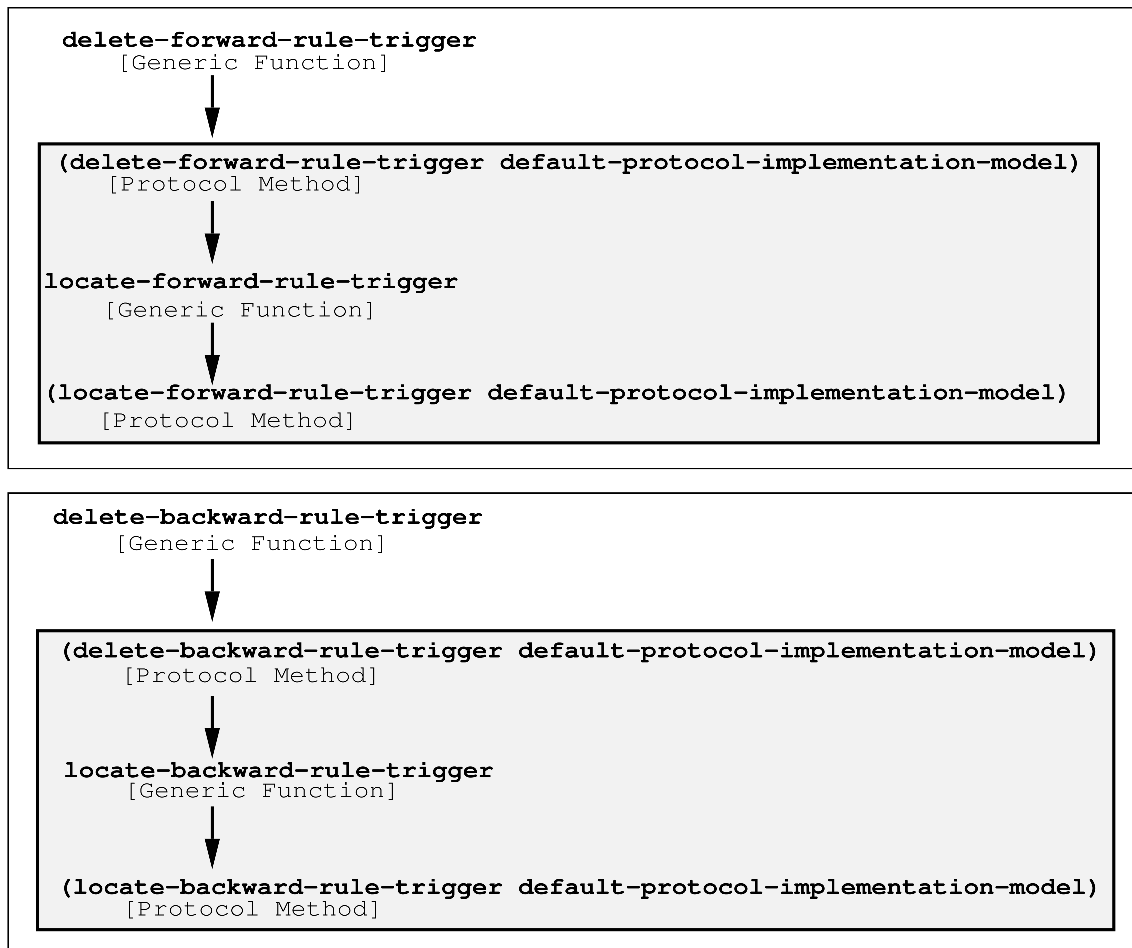


Figure 18. The Trigger-Deleting Protocol and Default Implementation

tures. The updating portion of the contract is implemented by a *continuation* argument to **joshua:locate-forward-rule-trigger**, **joshua:locate-forward-rule-trigger** and **joshua:locate-backward-question-trigger**.

This contract is implemented as follows:

- The locate method finds the place where the trigger is to be stored, or builds it, if it does not yet exist
- The method calls its continuation function, passing it the list of existing triggers that it just found.
- The continuation function:
    - Checks if the trigger being added (deleted) is new, or if it is a variant of an existing trigger (one trigger may represent several variant patterns).

- ° Updates its list of triggers to reflect the addition or deletion of the trigger, if it is not a variant.
- ° Returns three values:
  1. A new list of triggers (or the old one, if nothing has changed).
  2. A boolean flag indicating whether or not it modified the list of triggers
  3. The *canonical trigger* for this pattern. If the trigger being added is a variant of an existing trigger, then the existing trigger will be returned as the canonical trigger. If the trigger being inserted is the first such pattern, then it will be returned as the canonical trigger.

- The locate method updates the trigger index structure if the continuation indicates that a change is appropriate.
- The locate method returns the canonical trigger as its value.

Figures 17 and 18 show the trigger locating protocol and its default implementation.

Please consult the dictionary entries for the generic functions **joshua:locate-backward-rule-trigger**, **joshua:locate-forward-rule-trigger** and **joshua:locate-backward-question-trigger** for more detailed information about this protocol.

3.5.4. The Contract of the Trigger Mapping Functions

The trigger mapping protocol functions, **joshua:map-over-forward-rule-triggers** and **joshua:map-over-backward-rule-triggers** are responsible for looking up rule triggers for **joshua:tell** and **joshua:ask** in the place where the indexing functions have stored these triggers. The mapping functions walk over all triggers that might unify with the **joshua:tell** or **joshua:ask** pattern, and call the continuation on each candidate trigger. If you are writing your own trigger storage methods, your implementation of the trigger mapping functions must be consistent with the implementation of the trigger adding, locating, and deleting functions. See the section "Customizing the Rule Index", page 88.

3.5.4.1. Finding Forward Rule Triggers

When **joshua:tell** installs a new fact into the database, the system must find all forward rules that can be (partially) triggered by this new fact. It is the contract of **joshua:map-over-forward-rule-triggers** to look up the appropriate rule triggers.

In the default implementation, forward rule triggers are Rete Network nodes built by the compiler. The section "Forward Rule Triggers: the Rete Network", discusses this topic in more detail.

As we see from figure 19, finding the list of triggers happens in the course of justifying the newly inserted fact.

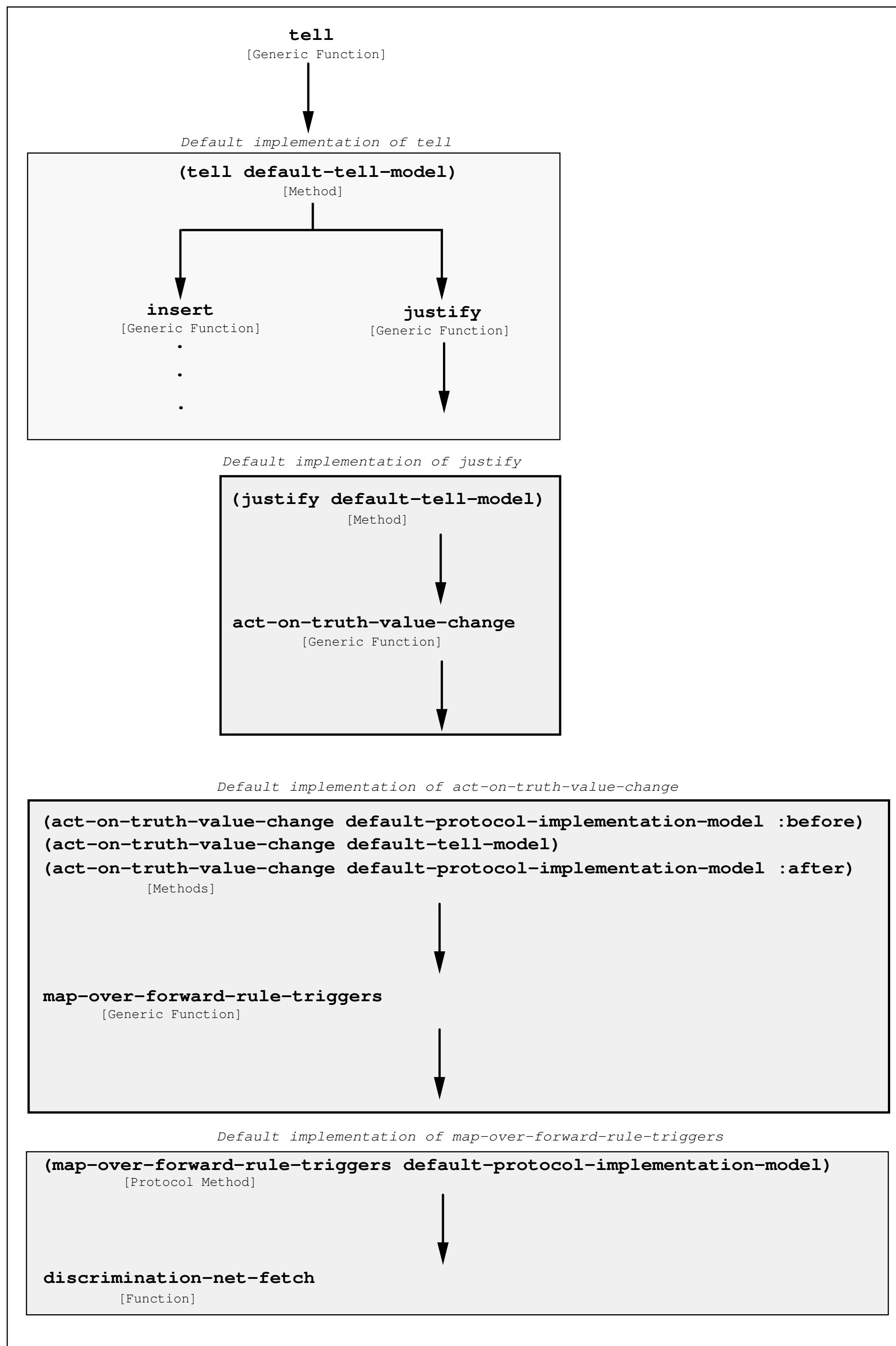


Figure 19. The **justify** protocol and its default implementation

The sequence is as follows:

The **joshua:tell** method forces the truth value of the just inserted predication to be **joshua:*unknown*** if the predication is new (not a variant of an existing fact). The **joshua:tell** method then calls **joshua:justify**.

Using the predication's current truth value of **joshua:*unknown***, its original truth value in the **joshua:tell** statement, and its justification as given in the keyword argument to **joshua:tell**, **joshua:justify** is responsible for:

- Setting the correct truth value for the predication
- Notifying the TMS (if one is being used) to propagate this truth value, and to make the current world consistent with this value.

The default **joshua:justify** method (which assumes that no TMS is being used) implements this contract as follows:

- Forces the truth value to correspond to the value passed to the method in an argument
- If this value differs from the current value of the predication (which is still **ju::unknown**), the **joshua:justify** method calls **joshua:*unknown***), the **joshua:justify** method calls **joshua:notice-truth-value-change** and **joshua::act-on-truth-value-change.**

joshua:act-on-truth-value-change has a primary method, and **:before** and **:after** methods. The primary method does nothing and can be overidden by the user, while the **:after** method does some internal bookkeeping for the TMS. (This is also true of the **joshua:notice-truth-value-change** protocol function). The **:before** method is the one of interest. It calls **joshua:map-over-forward-rule-triggers** and empties the forward rule queue.

The **joshua:map-over-forward-rule-triggers** method calls **joshua:discrimination-net-fetch** to get the applicable triggers. The continuation argument to the mapping function performs the unification if it is called.

For more on the justification protocol: See the section "The Truth Maintenance Protocol", page 54.

3.5.4.2. Finding Backward Rule Triggers

When you use **joshua:ask** to satisfy a goal, Joshua first looks in the database and then tries to run applicable backward rules and questions.

The protocol function **joshua:ask-rules** is the component of **joshua:ask** that finds backward rules to run, and that empties the backward rule queue. (**joshua:ask-data** tries to find database facts to satisfy the goal, and **joshua:ask-questions** tries to find and run applicable questions. See the section "The Contract of the Generic Functions **joshua:ask-data** and **joshua:fetch**", page 10. See the section "The Contract of the Trigger Locating Functions", page 39.)

joshua:ask-rules calls **joshua:map-over-backward-rule-triggers** to find appropriate backward rule triggers. Figure 20 shows the **joshua:ask-rules** protocol and default implementation.

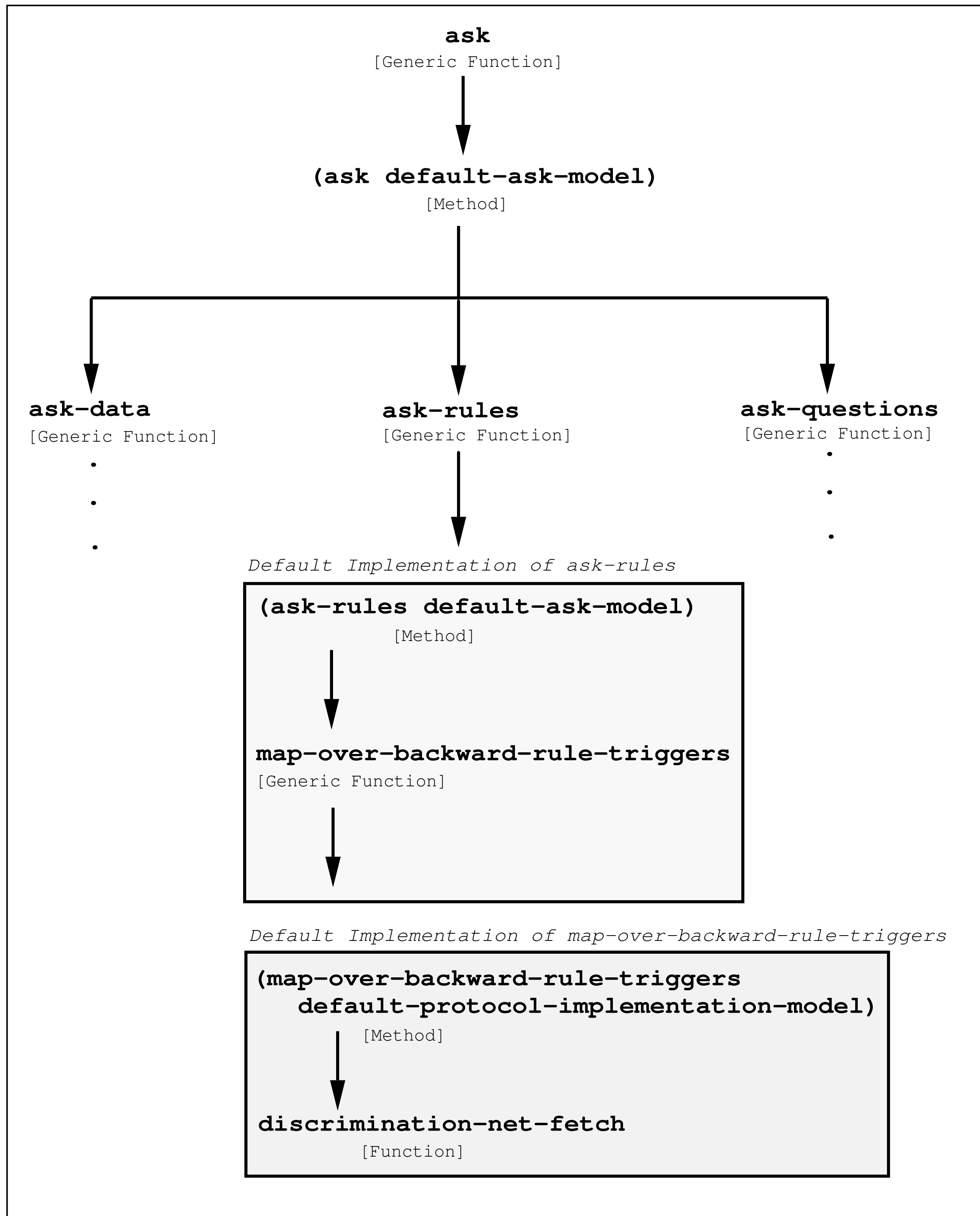


Figure 20. The ask-**rules** protocol and its default Implementation

joshua:map-over-backward-rule-triggers searches the index of backward rule triggers to find backward rules that can solve the goal. This function works analogously to the data-indexing function **joshua:fetch** that gets database facts for **joshua:ask-data**. Unification is done inside the rule; if unification succeeds, the rule performs the actions in the *if*-part.

The default rule index stores trigger information in a discrimination net. The default **joshua:map-over-backward-rule-triggers** method thus uses **joshua:discrimination-net-fetch** to search for backward triggers.

4. The Joshua Question Facilities

The basics of question syntax and operation were presented earlier: See the section "Asking the User Questions" in *User's Guide to Basic Joshua*. Here we elaborate a bit on ways of controlling question invocation. See the section "Controlling Question Invocation", page 47.

The bulk of the chapter discusses *question indexing*, that is, the way Joshua adds, deletes, and finds questions. This material is primarily useful if you want to provide your own implementation of these operations. If you are using the default question indexing, the topics discussed here are probably of no immediate interest.

4.1. Controlling Question Invocation

Typically during an **joshua:ask** the database is searched first (**joshua:fetch**), after which the appropriate rule trigger-mapping function (forward or backward) is executed, to find and run relevant rules. As a last step, question trigger-mapping functions are executed, to find and run backward questions (if **:do-questions** was set to non-**nil**).

Six built-in flavors for predicates used in **joshua:ask** goals are available to let you modify the above sequence. These flavors do subsets of what **joshua:ask** normally does, by leaving out one or more of the steps **joshua:ask-data**, **joshua:ask-rules**, or **joshua:ask-questions**. Thus the models save a certain amount of overhead when their predicates are used as goals to **joshua:ask**. The steps which *are* done are indicated by the names:

- **joshua:ask-data-only-mixin**
- **joshua:ask-rules-only-mixin**
- **joshua:ask-questions-only-mixin**
- **joshua:ask-data-and-rules-only-mixin**
- **joshua:ask-data-and-questions-only-mixin**
- **joshua:ask-rules-and-questions-only-mixin**

4.2. The Joshua Question Indexing Protocol

Joshua manages questions by manipulating question triggers. There are four trigger operations, namely:

- Adding triggers: **joshua:add-backward-question-trigger**
- Deleting triggers: **joshua:delete-backward-question-trigger**
- Locating triggers: **joshua:locate-backward-question-trigger**
- Iterating over triggers: **joshua:map-over-backward-question-triggers**

Figure 21 shows how the question indexing facilities relate to each other.

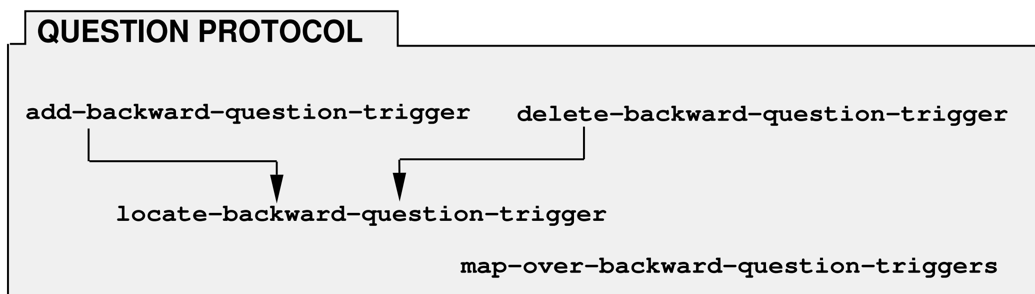


Figure 21. The Question Protocol

The contract of the question indexing functions is very similar to that of the data indexing functions. See the section "The Joshua Database Protocol", page 8.

4.2.1. The Contract of joshua:add-backward-question-trigger

The protocol function **joshua:add-backward-question-trigger** is analogous to the data-indexing function **joshua:insert**. When a new question is compiled, the compiler uses **joshua:add-backward-question-trigger** to add a trigger object to the data structure that holds trigger objects.

In the default implementation, the finding, building, and updating of trigger storage structures is the responsibility of the trigger locating function, **joshua:locate-backward-question-trigger**. Tailoring of backward question indexing is usually accomplished by providing methods for the **joshua:locate-backward-question-trigger** and **joshua:map-over-backward-question-triggers** functions.

Figure 22 shows the trigger adding protocol and its default implementation.

4.2.2. The Contract of joshua:delete-backward-question-trigger

joshua:undefquestion calls this protocol function with the pattern from the trigger part of a backward question. The function "unindexes" the trigger data structure of the backward question that corresponds to the pattern, making the question inaccessible.

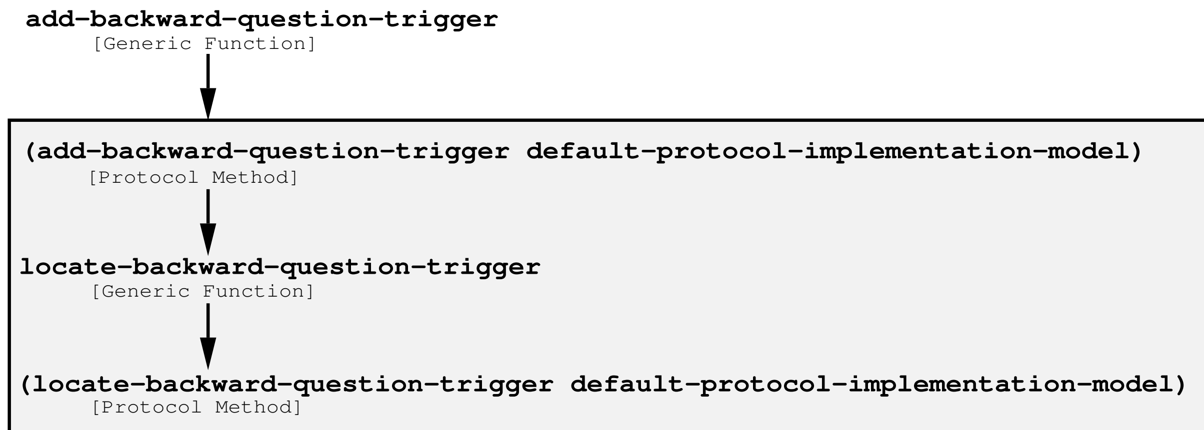


Figure 22. The Question Trigger Adding Protocol and Default Implementation

In the default implementation, the finding, building, and updating of trigger storage structures is the responsibility of the trigger locating function, **joshua:locate-backward-question-trigger**. Tailoring of backward question indexing is usually accomplished by providing methods for the **joshua:locate-backward-question-trigger** and **joshua:map-over-backward-question-triggers** functions.

Figure 23 shows the trigger deleting protocol and its default implementation.

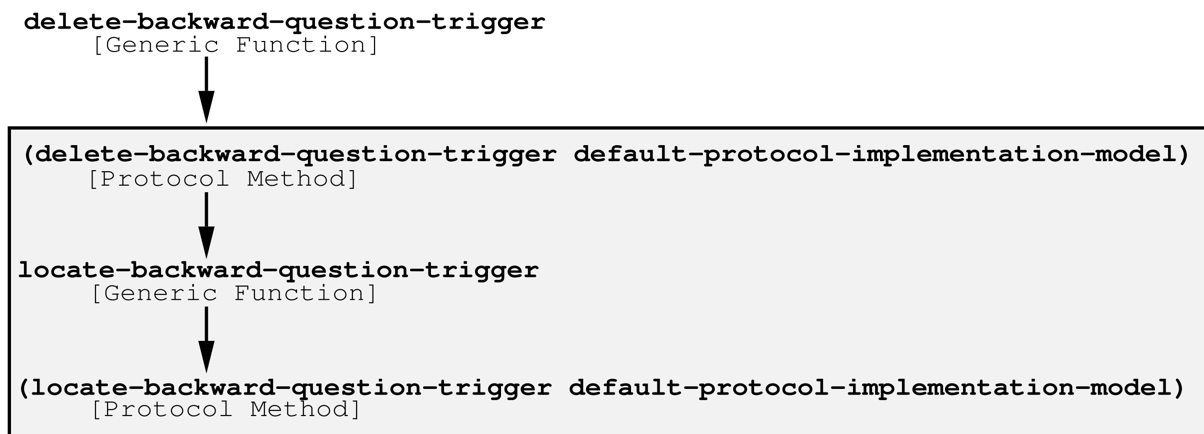


Figure 23. The Question Trigger Deleting Protocol and Default Implementation

4.2.3. The Contract of joshua:locate-backward-question-trigger

The **joshua:locate-backward-question-trigger** method is responsible for managing the data structures used to index backward question triggers. Each backward chaining question has a unique trigger structure, indexed by the pattern (and its truth value) of the question. Just as **joshua:insert** maps variant predications to a unique location in a data index, **joshua:locate-backward-question-trigger** locates

the unique place in a question index where Joshua stores a backward chaining question's trigger structure.

joshua:locate-backward-question-trigger is used as a subroutine of both **joshua:add-backward-question-trigger** and **joshua:delete-backward-question-trigger**. Knowledge of how to index a pattern is localized in the **joshua:locate-backward-question-trigger** methods, while the knowledge of the internal structure of the backward trigger data structures is localized in **joshua:add-backward-question-trigger** and **joshua:delete-backward-question-trigger**. These two higher levels routines call **joshua:locate-backward-question-trigger** passing to it *continuation*, a function which understands how to manipulate sets of backward question trigger data structures.

For more details see the section "The Contract of the Trigger Locating Functions", page 39.

4.2.4. The Contract of joshua:map-over-backward-question-triggers

joshua:map-over-backward-question-triggers is responsible for looking up backward question triggers capable of satisfying a query given to **joshua:ask**. It searches the questions index to find a set of backward question triggers whose patterns might unify with *predication* (the query given to **joshua:ask**), and calls *continuation* once for each backward question trigger found, thereby invoking the question.

If you are writing your own trigger storage methods, your implementation of the trigger mapping function must be consistent with the implementation of the trigger adding, deleting, and locating functions.

joshua:map-over-backward-question-triggers is the dual protocol function to **joshua:locate-backward-question-trigger**.

Figure 24 shows the **joshua:map-over-backward-question-triggers** protocol and default implementation.

4.2.4.1. Finding Backward Question Triggers

The protocol function **joshua:ask-questions** is the component of **joshua:ask** that finds backward questions to run, and that empties the backward question queue. (**joshua:ask-data** tries to find database facts to satisfy the goal, and **joshua:ask-rules** tries to find and run applicable rules.

See the section "The Contract of the Generic Functions **joshua:ask-data** and **joshua:fetch**", page 10. See the section "Finding Backward Rule Triggers", page 43.)

Figure 24 shows the **joshua:ask-questions** protocol and default implementation.

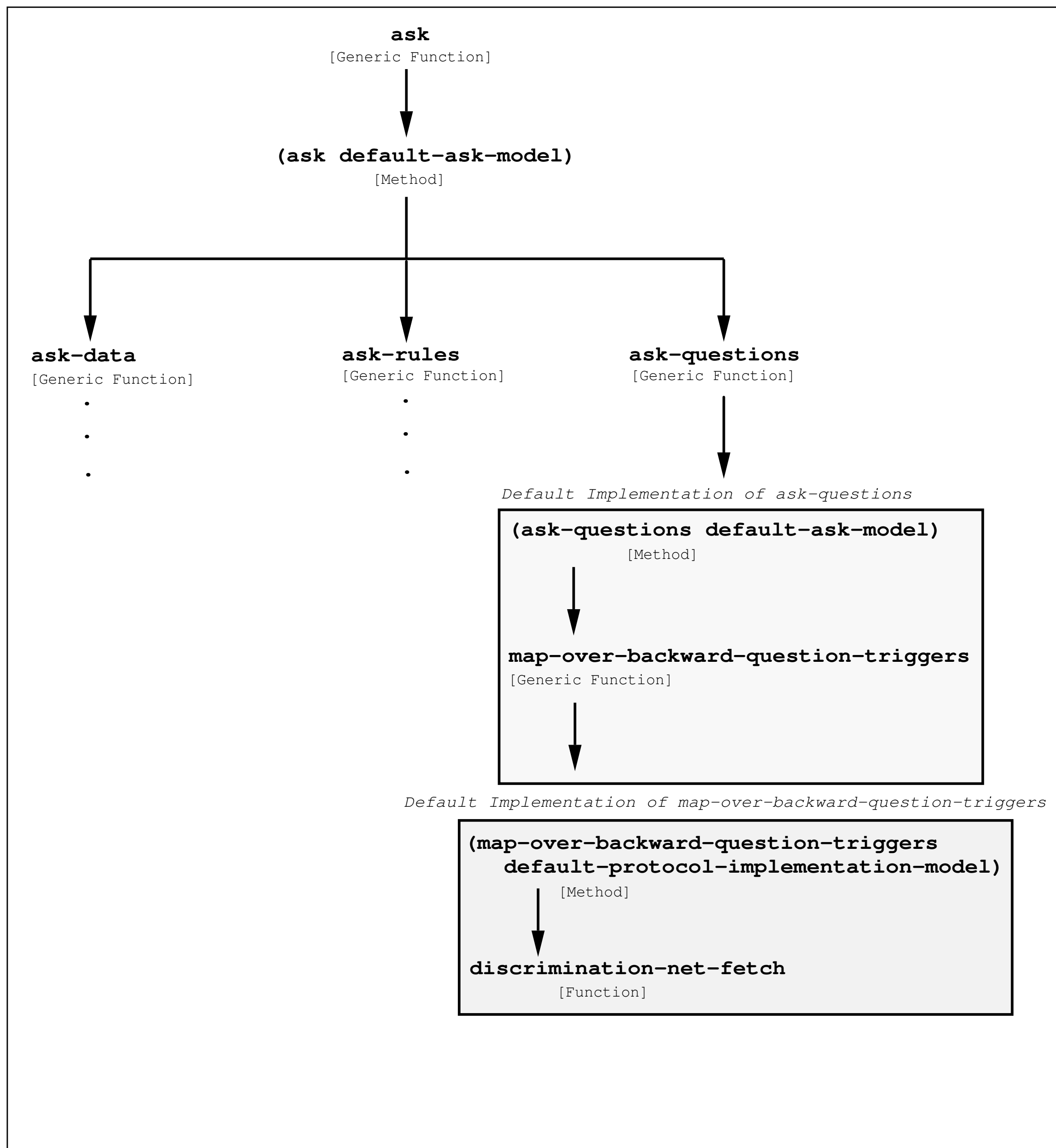


Figure 24. The **ask-questions** protocol and its default implementation

5. Truth Maintenance Facilities

We have covered the basic information about Truth Maintenance earlier: See the section "Justification and Truth Maintenance" in *User's Guide to Basic Joshua*.

This chapter provides a detailed explanation of the Truth Maintenance part of the Joshua protocol. It also explains how a different Truth Maintenance System (TMS) of your own design can be interfaced to Joshua. Finally, it provides a detailed explanation of the TMS supplied with Joshua.

(If you are interested in interfacing a new TMS of your own design to Joshua this chapter will provide useful information to you, however the information provided may not be sufficient. If you do wish to interface a TMS to Joshua, we strongly advise that you contact the Symbolics Consulting Services for assistance in building the interface.)

The Funtions of a Truth Maintenance System

A Truth Maintenance System performs several useful functions for Joshua:

1. The TMS is responsible for maintaining a record of why predications are believed to be true or false (hence the name Truth Maintenance); these records are called justifications.

2. The TMS can use these justifications to explain the reason why a predication is currently believed to be true (or false). As a special case of this, the TMS can identify the primitive beliefs (i.e. assumptions and premises) that are the ultimate reason for believing the predication.

3. The TMS can consistently propagate changes in truth-values. For example, suppose that the sole reason why predication B is believed to be true is that it was deduced from predication A. If A should ever change its truth-value to **joshua:*unknown***, then the TMS should also change the truth-value of B to **joshua:*unknown***. Similarly, if A should ever change its truth-value back to **joshua:*true***, then B should have its truth-value restored to **joshua:*true***.

4. It can consistently remove a justification from a predication. If this justification is the sole reason why the predication is believed, then the TMS must change the truth value of the predication to **joshua:*unknown*** and propagate the change of truth-value.

5. Finally, the TMS is responsible for ensuring that the database does not contain a contradiction. Whenever both a fact and its negation are asserted to be true, it is the TMS's job to determine what primitive beliefs (i.e. assumptions and premises) are ultimately responsible for the contradictory beliefs. The

TMS can then inform Joshua's error handlers of the situation by signalling a **joshua:tms-contradiction** condition. The handler which handles the condition, may then chose to unjustify one of the primitive beliefs underlying the contradiction in the hopes of removing the contradiction.

Types of Truth Maintenance Systems

There are several different varieties of Truth Maintenance Systems and these differ along several dimensions. Some TMS's such as Johan deKleer's ATMS maintains several viewpoints concurrently. The LTMS provided as a default in Joshua provides a single viewpoint at any one time, but allows you freely to switch back and forth between these viewpoints. Both of these styles of TMS have unique advantages and neither is appropriate in all circumstances.

Another dimension along which TMS's differ is whether they allow *non-monotonic justifications* in which one statement is believed because another fact has **joshua:*unknown*** truth-value. Jon Doyle's TMS supported this capability. The LTMS does not directly support this capability but allows it to be simulated using the **joshua:notice-truth-value-change** and the **joshua:act-on-truth-value-change** protocol methods.

A third dimension along which TMS's vary is whether their justification structures are unidirectional or multidirectional. Many TMS's (such as Doyle's and deKleer's) use a justification structure in which there is a unique conclusion and several antecedents. When all of the antecendents achieve their desired truth-values, the conclusion's truth-value is changed to that indicated by the justification. In effect, these TMS's perform only the simplest logic inference, namely *modus ponens*.

The LTMS provided with Joshua (and the 3-valued TMS of David McAllester upon which it is based) is a multidirectional TMS. In the LTMS, constituents of a justification are not restricted to playing a unique role as consequent or antecedent. Instead, the LTMS will change the truth-value of any constituent of the justification whenever the truth-values of all the other constituents force this choice.

5.1. The Truth Maintenance Protocol

The Joshua protocol provides a uniform mechanism for interfacing a TMS of your own design if the one supplied with Joshua does not meet your needs. (The current version of Joshua will not completely support a multiple-viewpoint TMS such as the ATMS, because of a difficultly of interfacing the triggering method for forward-chaining rules — the Rete Network — with the ATMS. This will be resolved in a subsequent release of Joshua. If you need this capability now, Symbolics personnel can help you figure out how to build the appropriate interface.)

5.1.1. The Contract of the Joshua TMS Protocol Functions

The interface between Joshua and a TMS consists of several Joshua protocol functions:

- **joshua:nontrivial-tms-p**: A protocol method that should return **t** for any predication which uses a TMS. This method is supplied by the predicate-model **basic-tms-mixin** which should be mixed into any predicate model that implements a TMS.

- **joshua:justify**: The protocol method that is used to tell the TMS to add a new justification. (This is most often invoked indirectly by providing a **:justification** argument to **joshua:tell**).

- **joshua:unjustify**: The protocol method that is used to tell the TMS to remove a justification from a predication.

- **joshua:current-justification**: The protocol method that is used to ask a predication for the justification that is responsible for its current truth-value. If the predication has **joshua:*unknown*** truth-value this should return **nil**.

- **joshua:all-justifications**: Returns all justifications into which the predication enters either as a supporting predication or as the predication supported by the justification.

- **joshua:notice-truth-value-change**: The protocol method used by the TMS to tell the rest of Joshua to update internal data structures to reflect the fact that a predication has changed its truth-value.

- **joshua:act-on-truth-value-change**: The protocol method used by the TMS to tell the rest of Joshua that a predication has changed its truth-value. In response, other parts of the application may initiate new inferential processes or produce visible side effects (such as updating a display).

5.1.2. The Contract of a Joshua TMS Justification

The TMS protocol functions allow Joshua to tell the TMS to create justifications. The format of these justifications is left completely up to the designer of the TMS (in the LTMS, justifications are implemented as *clauses* See the section "Clause Justification Structures", page 65. However, it is necessary for Joshua to be able to understand some of the information contained in a Justification, however it is implemented. Justifications are therefore required to obey a simple contract; they must be able to *destructure* themselves into several parcels of information, defined by the Joshua protocol.

Justifications used in a Joshua TMS must be understood by the generic function **joshua:destructure-justification**. (If justifications are implemented as flavor instances, this merely amounts to defining a **joshua:destructure-justification** method for the flavor of the justification. This is the approach used in the LTMS provided with Joshua).

Conceptually, every justification must contain the following information:

- **Mnemonic**: A name providing additional information, such as what rule created this justification or the type of a primitive justification (**:premise**, **:assumption**, and so on.)

- **Conclusion**: The predication supported by the justification.

- **True-support**: Those facts which must have truth-value **joshua:*true*** in order for the conclusion to follow.

- **False-support**: Those facts which must have truth-value **joshua:*false*** in order for the conclusion to follow.

- **Unknown-support**: Those facts which must have truth-value **joshua:*unknown*** in order for the conclusion to follow.

Justifications don't actually have to contain all this information; the **joshua:destructure-justification** generic function simply must return a value for each of these items. A TMS (such as the LTMS) which does not allow unknown-support does not actually have to have a field in the justification for this information, since it is uniformly empty. How the TMS stores information in a justification is completely at the discretion of the TMS implementor, as long as the protocol is obeyed.

5.1.3. TMS Utility Routines

Joshua provides a number of utility routines that will work with any TMS that obeys the protocol. Any predication that wants to use a TMS should mix in the Joshua predicate-model **joshua:basic-tms-mixin**; this provides default implementations for two protocol methods (**joshua:nontrivial-tms-p** and **joshua:support**). It also defines the **generation-mark** instance variable which is used by the default **joshua:support** method and may be useful for other TMS implementors. This instance variable can be used by any TMS which finds it convenient.

The utilities currently provided are:

- **joshua:support**: Return the primitive assertions which ultimately underlie the belief in a predication. This is a protocol method which can be overridden by another TMS implementation, although it is unlikely that this would be desirable.

- **joshua:support-with-name:** Returns a subset of the primitive assertions underlying a predication's belief. Only those predications with a justification whose Mnemonic is the second argument to this function are returned.

- **joshua:assumption-support**: Returns that subset of the primitive support of a predication which are justified by a justification whose Mnemonic is **:assumption**.

- **joshua:premise-support**: Returns that subset of the primitive support of a predication which are justified by a justification whose Mnemonic is **:premise**.

- **joshua:explain**: Prints an explanation of why a predication is believed to hold its current truth-value. This routine walks back through the tree of justifications that support a fact, printing one level of explanation for each level of justification.

- **joshua:remove-justification**: Takes a justification as argument and removes it from the Joshua world. This is a convenient function in some contexts; it is defined trivially in terms of **joshua:current-justification**, **joshua:unjustify** and **joshua:destructure-justification**.

- **joshua:graph-tms-support**: Takes a set of predications as arguments. Produces a graph display of the TMS justification structures supporting these predications. This graph continues backward until reaching predications which have primitive justifications.

An implementor of a TMS might need to use a few bits as flags as part of the internal algorithms of the TMS. These can of course be provided as instance variables that are part of the TMS mixin. However a few single bit flags are provided in all predications which may be accessed as `(TMS-Bits (Predication-Bits Predication))`.

5.1.4. Signalling Contradictions and Managing Backtracking

When a TMS detects a contradiction it must signal a condition See the section "Signalling Conditions" in *Symbolics Common Lisp Programming Constructs*. The condition signalled should be an instance of a flavor based on **joshua:tms-contradiction**. All such conditions should contain at least the following information:

- **The Contradictory Predication**: If the contradiction is detected by the TMS in such a way that it can localize the blame completely in an individual predication then this field should contain that predication. Some TMS's provide an entrypoint through which the user can declare a particular predication to be unacceptable even though it does not have **joshua:*contradictory*** truth-value (in the LTMS provided with Joshua this is called **ltms:backtrack**). If such is the case this field should contain the predication so blamed. This field is called **joshua:tms-contradiction-contradictory-predication**.

- **The Unsatisfiable Justification**: In some TMS's (in particular the LTMS provided with Joshua) contradictions are detected because a justification becomes unsatisfiable. In such a case this field should contain this invalid justification. If the user has declared a particular predication to be unacceptable and that predication has a current justification, then that justification should be included in this field. This field is called **joshua:tms-contradiction-justification**.

- **All the primitive support**: Given an unacceptable predication or an unsatisfiable justification (i.e. a contradiction), the TMS must determine the set of primitively supported predications that are in the support tree of the contradiction. Primitively supported predications are those whose justifications involve no other predication. This field is called **joshua:tms-contradiction-support**.

- **The subset of this which are premises**: A TMS may make a distinction between predications that may be retracted and those which are considered "immutable laws of the universe". This field of the condition should contain the subset of the primitive support which is considered unretractable, i.e. premises. This field is called **joshua:tms-contradiction-premises**.

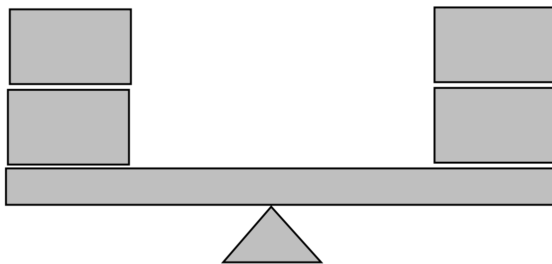
- **The subset of this which aren't premises**: This field of the condition signalled should contain that part of the primitive support which are allowed to be retracted. This field is called **joshua:tms-contradiction-non-premises**.

There is a default handler for the **joshua:tms-contradiction** condition provided in Joshua which handles two special cases automatically. If the condition signalled contains exactly one member of the non-premise primitive support set, then the handler automatically retracts this single predication (i.e. it removes its current justification). If the condition contains no non-premise primitive support, then the default handler signals another condition which should be based on **joshua:tms-hard-contradiction**. The intent is that this condition is one that a user might want to consider really wrong, so we provide a specific condition for this case. For the default handler to know what condition to signal, the first condition must implement a method for the **joshua:tms-contradiction-hard-contradiction-flavor** generic function; this must return the name of the flavor to be signalled for a **joshua:tms-hard-contradiction**. (For example, see the beginning of joshua:code;ltms).

By condition binding either or both of these conditions a program can completely control the backtracking process.

5.1.4.1. Using TMS Conditions: a Balance Beam Example

Suppose that we were writing a planning system for a blocks world construction task that includes balance beams like the following:



During the task of constructing a configuration like the one above, we must be careful that we never unbalance the beam enough to tip over the whole configuration. One situation that might result in such an unbalance is if we place a block

on one side without a counter balancing block at the other end. Another dangerous situation results when one block is grossly outweighed by a block at the other side. We'll call the first situation BLOCK-UNBALANCED-BY-BROTHER and the second BLOCK-OVERBALANCED-BY-BROTHER. The following are the predicates we use to describe this domain:

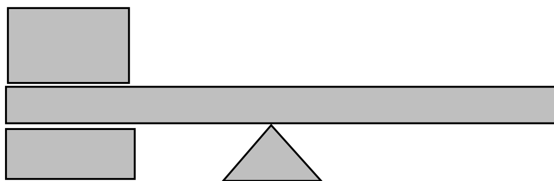
```
(define-predicate on (block balance-beam position) (ltms:ltms-predicate-model))
(define-predicate weight (block weight) (ltms:ltms-predicate-model))
(define-predicate block-unbalanced-by-brother (block supporter)
         (backtrack-when-true-mixin ltms:ltms-predicate-model))
(define-predicate block-overbalanced-by-brother (block supporter other-block)
                  (backtrack-when-true-mixin ltms:ltms-predicate-model))
```

Notice that BLOCK-OVERBALANCED-BY-BROTHER and BLOCK-UNBALANCED-BY-BROTHER both mix in the BACKTRACK-WHEN-TRUE-MIXIN. This flavor defines a **joshua:notice-truth-value-change** method that signals a contradiction if the predication it is mixed into ever becomes **joshua:*true***.

```
(define-predicate-model backtrack-when-true-mixin () ()
  (:required-flavors ltms:ltms-mixin))

(define-predicate-method (act-on-truth-value-change backtrack-when-true-mixin)
                                    (ignore)
 (when (eql (predication-truth-value self) *true*)
  (ltms:backtrack self)))
```

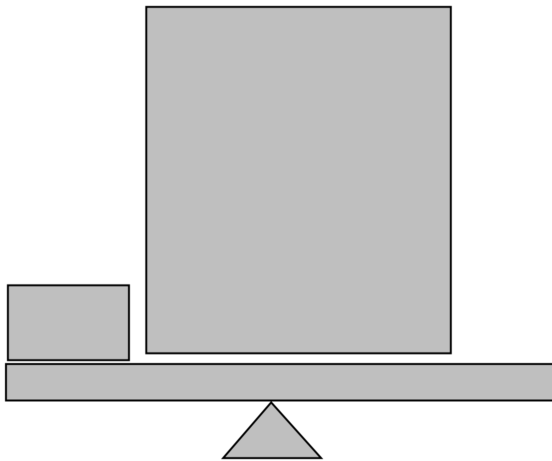
See the section "Notifying the LTMS of Contradictions", page 70.

When such a condition is signalled, we want to take recovery actions, by adding blocks that will keep the beam from tipping over. There are two such techniques. The first is a scaffold:



The second is a center weight:
The following predicates and rules describe and reason about these techniques:

```
(define-predicate scaffold (block balance-beam position)
        (ltms:ltms-predicate-model))
(define-predicate is-scaffolded (block position) (ltms:ltms-predicate-model))
(define-predicate is-counterweighted (block) (ltms:ltms-predicate-model))

(defrule detect-scaffolding (:forward)
 If [scaffold ?block ?supporter ?position]
 then [is-scaffolded ?supporter ?position])
```



```
(defrule detect-counterweighting (:forward)
 If [and [on ?block ?supporter center]
          [weight ?block very-heavy]]
 then [is-counterweighted ?supporter])

(defun is-counterweighted (balance-beam)
 (let ((counterweighted nil))
      (map-over-database-predications
       `[is-counterweighted ,balance-beam]
       #'(lambda (ignore) (setq counterweighted t)))
      counterweighted))

(defun is-scaffolded (balance-beam position)
 (let ((scaffolded nil))
      (map-over-database-predications
       `[is-scaffolded ,balance-beam ,position]
       #'(lambda (ignore) (setq scaffolded t)))
      scaffolded))
```

```
(defrule detect-unbalance (:forward)
  if [and [on ?block ?supporter left] :support ?f1
          [on nothing ?supporter right] :support ?f2]
  then (unless
         (or (is-counterweighted ?supporter)
             (is-scaffolded ?supporter 'left))
         (let ((missing-assumptions
                 (list (tell [not [is-counterweighted ?supporter]]
                             :justification :assumption)
                       (tell [not [is-scaffolded ?supporter left]]
                             :justification :assumption))))
           (tell [block-unbalanced-by-brother ?block ?supporter]
                 :justification `(unbalanced
                                  (,?f1 ,?f2)
                                  ,missing-assumptions
                                  nil))
           )))

(defrule detect-overbalance (:forward)
  if [and [on ?block-1 ?supporter left] :support ?f1
          [weight ?block-1 light] :support ?f2
          [on ?block-2 ?supporter right] :support ?f3
          [weight ?block-2 heavy] :support ?f4
          ]
  then (unless (is-scaffolded ?supporter 'right)
         (let ((missing-assumption
                 (tell [not [is-scaffolded ?supporter right]]
                       :justification :assumption)))
           (tell [block-overbalanced-by-brother ?block-1 ?supporter ?block-2]
                 :justification `(Overbalance
                                  (,?f1 ,?f2 ,?f3 ,?f4)
                                  (,missing-assumption))))))
```

The rules DETECT-OVERBALANCE and DETECT-UNBALANCE are responsible for noticing situations in which the balance beam will fall over. When one of these rule notices such a situation it causes a **joshua:tms-contradiction** to be signalled by **joshua:tell**ing a BLOCK-OVERBALANCED-BY-BROTHER or a BLOCK-UNBALANCED-BY-BROTHER predication (in effect, the rule "gripes" about the situation to use the term used in Scott Fahlman's BUILD program).

The application can respond to these "gripes" by binding a condition handler that rectifies the problem (Falhman used the name "gripe catcher" for the equivalent functionality in BUILD). See the section "Introduction to Signalling and Handling Conditions" in *Symbolics Common Lisp Programming Constructs*.

Here are two functions that can be used as "gripe catchers":

```
(defun handle-overbalanced-condition (condition-object)
  (let ((contradictory-predication
          (tms-contradiction-contradictory-predication condition-object)))
    (if (typep contradictory-predication 'block-overbalanced-by-brother)
        (let ((no-scaffolding
                (find 'is-scaffolded
                      (tms-contradiction-non-premises condition-object)
                      :key #'(lambda (thing)
                               (predication-predicate
                                 (multiple-value-bind (ignore supportee)
                                     (destructure-justification thing)
                                   supportee))))))
          (with-statement-destructured (light-guy supporter heavy-guy)
                                       contradictory-predication
            (format t "~&Overbalance of ~s on ~s by ~s noticed"
                    light-guy supporter heavy-guy)
            (remove-justification no-scaffolding)
            (tell `[scaffold ,(gentemp "BLOCK-") ,supporter right]
                  :justification :premise)
            t))
        (values))))

(defun handle-unbalanced-condition (condition-object)
  (let ((contradictory-predication
          (tms-contradiction-contradictory-predication condition-object)))
    (if (typep contradictory-predication 'block-unbalanced-by-brother)
        (let ((no-counterweight
                (find 'is-counterweighted
                      (tms-contradiction-non-premises condition-object)
                      :key #'(lambda (thing)
                               (predication-predicate
                                 (multiple-value-bind (ignore supportee)
                                     (destructure-justification thing)
                                   supportee))))))
          (with-statement-destructured (light-guy supporter)
                                       contradictory-predication
            (format t "~&Unbalance of ~s on ~s"
                    light-guy supporter)
            (remove-justification no-counterweight)
            (let ((new-block (gentemp "BLOCK-")))
              (tell `[on ,new-block ,supporter center]
                    :justification :premise)
              (tell `[weight ,new-block very-heavy]
                    :justification :premise)))
          t)
        (values))))
```

Each of these fetches the contradictory predication from the condition object, and

then checks that it is the type of gripe which this function wants to handle. If so, it examines the assumption support part of the condition object. For example, HANDLE-UNBALANCED-CONDITION looks for an IS-COUNTERWEIGHTED statement (which the handler assumes has truth-value **joshua:*false***) in the assumption support. [Notice that the condition object contains the *justifications* of the assumptions underlying the contradiction. So the handler must destructure the justification to get the assumption *predication*. It then tests if the *predicate* of the predication is IS-COUNTERWEIGHTED]. If the condition object represents a situation that HANDLE-UNBALANCED-CONDITION can manage, it then repairs things by **joshua:unjustify**ing the assumption that there is no counterweight (using **joshua:remove-justification**) and then **joshua:tell**ing two new statements: the first states that there is a block on the center of the balance beam; the second states that the block is very heavy. In effect the condition handler, repairs the situation by making there be a heavy centerweight. The other handler repairs the situation where there is a single block at one end, by inserting a scaffold under the beam. Each of the handlers follows the usual protocol for condition handlers of return **joshua::t** if it handled the condition and **joshua::nil** if it declines to handle the condition.

One can now use these handlers by **joshua::condition-bind**ing them and then describing a world situation (or running a planner). For example:

```
(condition-bind ((tms-contradiction #'handle-overbalanced-condition))
  (condition-bind ((tms-contradiction #'handle-unbalanced-condition))
    (clear)
    (tell [on block-1 balance-beam left] :justification :assumption)
    (tell [weight block-1 light])
    (tell [on nothing balance-beam right] :justification :assumption)
    ))
```

The following figure shows a trace of this code from the point where the condition handler takes control.

5.1.5. Signalling Truth Value Changes

As a result of resolving a contradiction, a TMS may cause the truth-value of many facts to change. A contradiction is usually resolved by unjustifying some member of its primitive support; this causes the unjustified predication to change from a definite truth-value (**joshua:*true*** or **joshua:*false***) to **joshua:*unknown***. Any predication which depended on the retracted one similarly changes its truth-value to **joshua:*unknown***. In some TMS's however, some facts may change from **joshua:*unknown*** to a definite truth-value.

It is the responsibility of the TMS to inform the rest of the Joshua application of these changes in truth-values. To do this the TMS should call the **joshua:notice-truth-value-change** and the **joshua:act-on-truth-value-change** methods for each fact which undergoes a transition in truth-values. The **joshua:notice-truth-value-change** method can then update any data structures that are maintained outside the TMS to correspond to the changed truth-values. In addition, the **joshua:act-on-truth-value-change** method may initiate a deductive process or otherwise affect the world.

```
Unbalance of BLOCK-1 on BALANCE-BEAM
   ▶ Unjustifying ¬[IS-COUNTERWEIGHTED BALANCE-BEAM]
   ▶ Telling predication [ON BLOCK-1730 BALANCE-BEAM CENTER]
   ▶ Justifying ?[ON BLOCK-1730 BALANCE-BEAM CENTER]
   ▶ Telling predication [WEIGHT BLOCK-1730 VERY-HEAVY]
   ▶ Justifying ?[WEIGHT BLOCK-1730 VERY-HEAVY]
   ▶ Changing truth value of ?[WEIGHT BLOCK-1730 VERY-HEAVY] from unknown to true
   ▶ Changing truth value of ?[ON BLOCK-1730 BALANCE-BEAM CENTER] from unknown to true
   ▶ Changing truth value of ¬[IS-COUNTERWEIGHTED BALANCE-BEAM] from false to true
   ▶ Changing truth value of [BLOCK-UNBALANCED-BY-BROTHER BLOCK-1 BALANCE-BEAM] from true to un
known
   ▶ Changing truth value of ?[WEIGHT BLOCK-1730 VERY-HEAVY] from unknown to true
   ▶ Changing truth value of ?[ON BLOCK-1730 BALANCE-BEAM CENTER] from unknown to true
     ▶ Telling predication [IS-COUNTERWEIGHTED BALANCE-BEAM]
     ▶ Justifying [IS-COUNTERWEIGHTED BALANCE-BEAM]
     ▶ Changing truth value of ¬[IS-COUNTERWEIGHTED BALANCE-BEAM] from false to true
     ▶ Changing truth value of [BLOCK-UNBALANCED-BY-BROTHER BLOCK-1 BALANCE-BEAM] from true to
unknown
     ▶ Changing truth value of ¬[IS-COUNTERWEIGHTED BALANCE-BEAM] from false to true
     ▶ Changing truth value of [BLOCK-UNBALANCED-BY-BROTHER BLOCK-1 BALANCE-BEAM] from true to
unknown
>Breakpoint ZMACS.  Press RESUME to continue or ABORT to quit.

⇨ :Show Joshua Database (matching pattern [default All]) All
True things
  [ON BLOCK-1730 BALANCE-BEAM CENTER]  [IS-COUNTERWEIGHTED BALANCE-BEAM]
  [ON NOTHING BALANCE-BEAM RIGHT]      [WEIGHT BLOCK-1730 VERY-HEAVY]
  [ON BLOCK-1 BALANCE-BEAM LEFT]       [WEIGHT BLOCK-1 LIGHT]
False things
  [IS-SCAFFOLDED BALANCE-BEAM LEFT]
⇨
```

Figure 25. Example Trace of Condition Handler

Notice that although there may be several predications that have changed truth-value, the **joshua:notice-truth-value-change** methods for these predications are invoked sequentially. As the method for each predication is called, it updates the current view of the world to correspond to the changed truth value of the predication. Thus, until all the methods have had a chance to run, the model of the world will be partially inconsistent.

To allow this problem to be addressed, there are two mehtods to handle the updating: **joshua:notice-truth-value-change** and **joshua:act-on-truth-value-change**. The TMS should first call the **joshua:notice-truth-value-change** method for every predication that has changed truth-value. After that, the TMS should call the **joshua:act-on-truth-value-change** method for each predication that has changed truth-value.

A predication's **joshua:notice-truth-value-change** method should update whatever internal data-structures require modification, but should avoid any other action that might depend on examining other data-structures which have not yet been updated. The **joshua:act-on-truth-value-change** method is allowed to take whatever actions it desires.

The most important example of this two-pass protocol is the Rete-Network used to trigger forward chaining rules. During the **joshua:notice-truth-value-change** pass, the Rete-Network updates its internal data structures to remove partial triggering information that depended on the previous truth-value of predications that have changed truth-value. (If this were not done, then rules might be executed even though the triggering predications no longer have the truth value appropriate for triggering the rule). During the **joshua:act-on-truth-value-change** pass the new truth-values of predications are propagated through the network, allowing rules

that have a valid triggering set to execute. TMS implementors do not have to concern themselves with these details, since they are implemented by system supplied **:before** and **:after** methods.

5.2. The Joshua LTMS

This section explains how the Truth Maintenance System provided with Joshua works and how to use its features. The Joshua TMS, which we call an LTMS (using terminology due to Forbus and deKleer) is derived from the 3-valued Truth Maintenance Sytem developed by David McAllester at MIT.

The Joshua LTMS also provides an additional feature (the :**One-Of** justification) which allows you to control the invocation of assumptions.

5.2.1. Clause Justification Structures

The justification structure used in the **LTMS** is called a *clause*. A clause is simply a logical disjunction of several facts. For example,

$$F_1 \vee F_2 \vee \ldots F_n$$

Logically, if all but one of these facts is false, then the other must be true. For example, if F_1, F_2, ... , F_{n-1} are all false, then F_n must be true. Similarly, if F_1, F_2, ... , F_{i-1},F_{i+1}, ... , F_n are all false, then F_i must be true. (This is known both as the *Cut* rule and as *Unit-Resolution*). Thus, a clause can be used to perform as many inferences as there are constituents of the clause.

The normal *modus ponens* rule is actually a special case of the above clausal inferencing. This is because

$$P \rightarrow Q$$

is logically equivalent to

$$\neg P \vee Q.$$

Once an implication has been converted to clausal form, the normal *modus ponens* rule follows immediately (since P is the negation of ¬P, every constituent of the clause ¬P ∨ Q but Q is false and, therefore Q must be true). In addition, clausal inferencing can deduce ¬P from ¬Q (since ¬Q negates Q leaving only ¬P).

The clausal mechanism used in the Joshua LTMS diverges from this simple description only slightly. First of all, in addition to its constituents a clause also containts a *mnemonic*. If the clause was created to memoize an inference drawn by a rule, then the mnemonic should be the name of the rule. In other cases, the mnemonic may be used to indicate some special property of the clause.

The second variation is that since in Joshua both P and ¬P are represented by the same database predication, it is necessary to indicate in the clause data structure which truth-value of P is intended. Thus, a clause actually consists of two lists of predications: the positive constituents and the negated constituents. The intended truth-value of the positive constituents is ***true*** and the intended truth-value of the negated constituents is ***false***.

If all but one of the constituents of a clause have the opposite truth-value from their intended truth-value, then the final constituent is forced to assume its intended truth-value. (Note: ***true*** and ***false*** are each other's opposite truth-value; ***unknown*** is not the opposite truth-value of either ***true*** or ***false***).

A unit (or primitive) justification in the LTMS is simply a clause with only one constituent. This single constituent will, therefore, be forced to assume its intended truth-value. We refer to the predications so justified as *primitively justified* predications. One special kind of primitively justified predication is a *premise*; these have a supporting clause whose mnemonic is **:premise**.

If every constituent of a clause has the opposite truth-value from its intended truth-value, then the clause is *unsatisfiable* and a contradiction is signalled. See the section "Signalling Contradictions and Managing Backtracking", page 57.

The Condition signalled by the LTMS is called **ltms:ltms-contradiction**. If the support underlying the contradiction contains only premises then the condition called **ltms:ltms-hard-contradiction** is signalled. These conditions have the same instance variables as the base **tms-contradiction** flavors.

Nogoods in the LTMS

When the LTMS signals a contradiction it automatically constructs a new clause, called a *nogood*. The idea behind the nogood is as follows: there is a set of primitively justified predications whose current truth-value assignments led to the contradiction. Since a contradiction is unacceptable, at least one of these primitively justified predications must have the opposite truth-value from that which it currently has.

Thus suppose we justify a fact with three justifications as follows:

```
(define-predicate loser (a) (ltms:ltms-predicate-model))
(define-predicate cause-of-lossage (a) (ltms:ltms-predicate-model))

(let  ((cause-1 (tell [cause-of-lossage a] :justification :assumption))
       (cause-2 (tell [cause-of-lossage b] :justification :assumption))
       (cause-3 (tell [cause-of-lossage c] :justification :assumption)))
 (tell [loser X]
        :justification `(causing-part-of-lossage-1
                            (,cause-1 ,cause-2 ,cause-3))))
```

The result is shown in figure 26:

```
 ▶ Telling predication [CAUSE-OF-LOSSAGE A]
 ▶ Telling predication [CAUSE-OF-LOSSAGE B]
 ▶ Telling predication [CAUSE-OF-LOSSAGE C]
 ▶ Telling predication [LOSER X]
[LOSER X]
T
```

Figure 26. Example of setting up a nogood clause

Then the primitive support of the fact will consist of the three assumptions as can be seen below:

```
(map-over-database-predications [loser X] #'explain)

[LOSER X] is true
  It was derived from rule CAUSING-PART-OF-LOSSAGE-1
  [CAUSE-OF-LOSSAGE A] is true
    It is an ASSUMPTION
  [CAUSE-OF-LOSSAGE B] is true
    It is an ASSUMPTION
  [CAUSE-OF-LOSSAGE C] is true
    It is an ASSUMPTION

(support [LOSER X])
([CAUSE-OF-LOSSAGE A] [CAUSE-OF-LOSSAGE B] [CAUSE-OF-LOSSAGE C])
```

If we now tell the LTMS that the predication `[LOSER X]` is contradictory then, in addition to invoking the contradiction handler, it will create a new Nogood clause:

```
#<LTMS:NOGOOD ¬[CAUSE-OF-LOSSAGE A]
              ∨ ¬[CAUSE-OF-LOSSAGE B]
              ∨ ¬[CAUSE-OF-LOSSAGE C]>
```

Which says that at least one of the causes of the contradiction must be ***false*** since each of them was true at the time the contradiction occured. A constituent of a clause is printed with a leading negation sign (¬) if its intended truth-value in the clause if ***false***.

A *Nogood* is just a normal clause whose *mnemonic* is **ltms:nogood**. Once created, it behaves no differently from any other clause. (However, nogoods are internals of the LTMS that need never be manipulated by a user).

Suppose that `[CAUSE-OF-LOSSAGE C]` was unjustified in order to resolve the above contradiction and that the above nogood was then installed. Notice that both `[CAUSE-OF-LOSSAGE A]` and `[CAUSE-OF-LOSSAGE B]` are still **joshua:*true***. Therefore, `[CAUSE-OF-LOSSAGE C]` is the only constituent of the nogood which does not have the opposite truth-value from the clause's intended truth-value. The LTMS will, therefore, force `[CAUSE-OF-LOSSAGE C]` to assume its intended truth-value of ***false***.

Controlling Choices in the LTMS

The LTMS recognizes one special type of clause called a *One-Of*. One-Ofs are distinguished by having a **mnemonic** field whose value is **:one-of**. One-ofs can be used to control the making of assumptions.

In a normal clause, the LTMS forces a constituent to assume a definite truth-value only when every other constituent has the opposite truth-value from that intended for it by the clause. Thus, if every constituent of a clause has ***unknown*** truth-value, the LTMS will take no action.

In a One-Of clause, however, the LTMS will guarantee that at least one constituent has its intended definite truth-value. It does this by adding to that constituent a primitive justification whose **mnemonic** is :**choice**.

```
⇨ (ltms:backtrack [LOSER X])
Error: Backtracking because:
       [LOSER X] is contradictory
       #<CAUSING-PART-OF-LOSSAGE-1 [CAUSE-OF-LOSSAGE A]  ^ [CAUSE-OF-LOSSAGE B] ^ [CAUSE-OF-LOSSAGE C] → [LOSER X]> i
s the unsatisfiable clause.
                    [CAUSE-OF-LOSSAGE C]
                       :ASSUMPTION
[LOSER X]           [CAUSE-OF-LOSSAGE B]
                       :ASSUMPTION
                    [CAUSE-OF-LOSSAGE A]
                       :ASSUMPTION

LTMS::PICK-AND-UNJUSTIFY-A-SUPPORTER
   Arg 0 (SUPPORT): (#<:ASSUMPTION [CAUSE-OF-LOSSAGE A]> #<:ASSUMPTION [CAUSE-OF-LOSSAGE B]>
                     #<:ASSUMPTION [CAUSE-OF-LOSSAGE C]>)
   Arg 1 (LTMS::CLAUSE): #<CAUSING-PART-OF-LOSSAGE-1 [CAUSE-OF-LOSSAGE A]  ^ [CAUSE-OF-LOSSAGE B] ^ [CAUSE-OF-LOSSAGE
 C] → [LOSER X]>
   Arg 2 (LTMS::ORIGINATOR): [LOSER X]
   Arg 3 (LTMS::CONDITION-BUILDER): NIL
s-A, RESUME:    Pick a subset of these predications to be unjustified.
s-B, ABORT:     Return to Lisp Top Level in Dynamic Lisp Listener 1
→ Resume Proceed
Pick a subset of these predications to be unjustified.
Subset to be unjustified: #<:ASSUMPTION [JU::CAUSE-OF-LOSSAGE JU::C]>
 ▶ Unjustifying [CAUSE-OF-LOSSAGE C]
 ▶ Changing truth value of [CAUSE-OF-LOSSAGE C] from true to false
 ▶ Changing truth value of [LOSER X] from true to unknown
 ▶ Changing truth value of [CAUSE-OF-LOSSAGE C] from true to false
 ▶ Changing truth value of [LOSER X] from true to unknown
```

Although you can create One-Of clauses directly (by using **justify** or the **:justification** keyword argument to **tell**) it is usually easier and more effective to use the special predicate **ltms:one-of** provided with the LTMS to do this, for example:

```
(tell [ltms:one-of [loser X] [loser Y] [loser Z]]
      :justification :assumption)
```

```
 ▶ Telling predication [LTMS:ONE-OF [LOSER X] [LOSER Y] [LOSER Z]]
 ▶ Justifying ?[LTMS:ONE-OF [LOSER X] [LOSER Y] [LOSER Z]]
 ▶ Changing truth value of ?[LTMS:ONE-OF [LOSER X] [LOSER Y] [LOSER Z]] from unknown to true
 ▶ Changing truth value of ?[LTMS:ONE-OF [LOSER X] [LOSER Y] [LOSER Z]] from unknown to true
 ▶ Telling predication [LOSER X]
 ▶ Telling predication [LOSER Y]
 ▶ Telling predication [LOSER Z]
 ▶ Justifying [LTMS:ONE-OF [LOSER X] [LOSER Y] [LOSER Z]]
 ▶ Justifying ?[LOSER X]
 ▶ Changing truth value of ?[LOSER X] from unknown to true
 ▶ Changing truth value of ?[LOSER X] from unknown to true
[LTMS:ONE-OF [LOSER X] [LOSER Y] [LOSER Z]]
T
⇨ :Show Joshua Database (matching pattern [default All]) All
True things
  [LTMS:ONE-OF [LOSER X] [LOSER Y] [LOSER Z]]
  [LOSER X]
False things
  None
```

Notice that this creates and inserts into the database one predication for each constituent of the One-Of; however, it initially provides no justification for any of these constituents. In addition, it creates a **ltms:one-of** predication which is justified as an **:assumption** as directed by the **joshua:tell**. Finally, since no constituent of the One-Of has its intended truth-value, the LTMS picks one and justifies it with a **:choice** justification.

This choice can be overridden by explicitly asserting its negation with a premise justification (as far as the LTMS is concerned, a **:choice** primitive justification is just an assumption; only **:premise** justifications are treated as unretractable):

```
(tell [not [loser x]] :justification :premise)
```

Notice that when the choice of `[LOSER X]` is overriden, the LTMS picks another constitutent of the clause to justify with a **:choice** justification. This leaves the

```
 ▶ Telling predication [NOT [LOSER X]]
 ▶ Telling predication [not [LOSER X]]
 ▶ Justifying [LOSER X]
 ▶ Unjustifying [LOSER X]
 ▶ Changing truth value of [LOSER X] from true to false
 ▶ Justifying ?[LOSER Y]
 ▶ Changing truth value of ?[LOSER Y] from unknown to true
 ▶ Changing truth value of ?[LOSER Y] from unknown to true
 ▶ Changing truth value of [LOSER X] from true to false
¬[LOSER X]
NIL
```

database in the following state:

```
 ➪ :Show Joshua Database (matching pattern [default All]) All
 True things
   [LTMS:ONE-OF [LOSER X] [LOSER Y] [LOSER Z]]
   [LOSER Y]
 False things
   [LOSER X]
```

Similarly, if we now override this choice, the following will result:

```
(tell [not [loser y]] :justification :premise)
```

```
 ▶ Telling predication [NOT [LOSER Y]]
 ▶ Telling predication [not [LOSER Y]]
 ▶ Justifying [LOSER Y]
 ▶ Unjustifying [LOSER Y]
 ▶ Changing truth value of ?[LOSER Z] from unknown to true
 ▶ Changing truth value of [LOSER Y] from true to false
 ▶ Changing truth value of ?[LOSER Z] from unknown to true
 ▶ Changing truth value of [LOSER Y] from true to false
¬[LOSER Y]
NIL
➪ :Show Joshua Database
True things
  [LTMS:ONE-OF [LOSER X] [LOSER Y] [LOSER Z]]
  [LOSER Z]
False things
  [LOSER Y]
  [LOSER X]
```

Finally, if we override the last choice, then the **ltms:one-of** predication will simply be retracted since it is justified as an **:assumption**:

```
(tell [not [loser z]] :justification :premise)
```

```
 ▶ Telling predication [NOT [LOSER Z]]
 ▶ Telling predication [not [LOSER Z]]
 ▶ Justifying [LOSER Z]
 ▶ Unjustifying [LTMS:ONE-OF [LOSER X] [LOSER Y] [LOSER Z]]
 ▶ Changing truth value of [LTMS:ONE-OF [LOSER X] [LOSER Y] [LOSER Z]] from true to fal
 ▶ Changing truth value of [LOSER Z] from true to false
 ▶ Changing truth value of [LTMS:ONE-OF [LOSER X] [LOSER Y] [LOSER Z]] from true to fal
 ▶ Changing truth value of [LOSER Z] from true to false
¬[LOSER Z]
NIL
```

And at this point the database will look like:

```
➪ :Show Joshua Database
True things
  None
False things
  [LTMS:ONE-OF [LOSER X] [LOSER Y] [LOSER Z]]
  [LOSER Z]
  [LOSER Y]
  [LOSER X]
```

Notifying the LTMS of Contradictions

The LTMS notices logical contradictions any time that a predication is about to assume both **joshua:*true*** and **joshua:*false*** truth-values. At such tims, the LTMS intervenes and initiates *backtracking* (the process of handling and removing contradictions). However, there are times when you may want the LTMS to treat some condition as if it's a contradiction, even though there is no predication which is contradictory.

The LTMS provides two techniques for doing this. The first of these is the function **ltms:backtrack**. The second is the **ltms:contradiction** predicate.

The function **ltms:backtrack** takes three arguments. The first should be a database predication, the second a truth-value that defaults to the current truth-value of the predication. The third argument is used to instruct the LTMS to signal a user-defined condition. Calling **ltms:backtrack** causes the LTMS to initiate backtracking just as if its first argument had become contradictory. Backtracking will continue until the predication has a truth-value other than the second argument to **ltms:backtrack**.

One technique for using this function is via the **joshua:notice-truth-value-change** protocol function. For example:

```
(define-predicate I-should-never-be-in () (ltms:ltms-predicate-model))

(define-predicate-method (act-on-truth-value-change I-should-never-be-in)
                                  (ignore)
 (when (eql (predication-truth-value self) *true*)
  (ltms:backtrack self)))

(tell [I-should-never-be-in] :justification :assumption)

  Telling predication [I-SHOULD-NEVER-BE-IN]
  Justifying: [I-SHOULD-NEVER-BE-IN] <-- ASSUMPTION
  Justifying: [I-SHOULD-NEVER-BE-IN] as false <-- NOGOOD

¬[I-SHOULD-NEVER-BE-IN]
T
```

Notice that as soon as the `[I-SHOULD-NEVER-BE-IN]` predication is justified as an assumption, the **joshua:notice-truth-value-change** method is invoked causing backtracking to begin. This creates a nogood which causes the predication to assume **joshua:*false*** truth-value.

The predicate **ltms:contradiction** is defined in a manner similar to the `I-should-never-be-in` predicate in the example above. Thus, it can be used to notify the LTMS of a contradiction. Whenever a **ltms:contradiction** predication assumes a truth-value of **joshua:*true*** backtracking is initiated. The following rule (which is in the Jericho system of Joshua examples) causes backtracking to be initiated whenever any type of tragedy is deduced.

```
(defrule trying-to-write-a-comedy (:forward)
  ;; no tragedies, please
  IF [tragedy ?fact]
  THEN [ltms:contradiction])
```

6. Joshua Metering

Joshua extends the system metering facilities in order to provide specific tools for analyzing Joshua programs. These tools are conveniently available in the Metering Interface. Joshua metering can help you do three things:

- Find bugs in your program that show up only as performance problems
- Improve the performance of your Joshua rules by changing the ordering of the triggers
- Use the Joshua modeling capabilities.

Because the Metering system is not part of the default world, you must load it separately, using the command:

```
Load System Joshua Metering
```

By loading the Joshua Metering system you also load the standard Metering system.

Before using Joshua metering you should familiarize yourself with how to use the Metering Interface. See the section "Metering Interface" in *Program Development Utilities*.

6.1. Joshua Metering Types

Joshua defines new Metering Types designed for metering Joshua programs. Unlike the system-provided metering types which collect data about function calls, the Joshua metering types collect data about the Joshua Protocol steps and the forward rule Rete network. This lets you study the execution of your Joshua programs without overwhelming you with the details of every function call.

There are three new Metering Types:

- Joshua Tell Metering
- Joshua Ask Metering
- Joshua Merge Metering

6.1.1. Joshua Tell Metering

The Joshua Tell Metering type collects information about each **joshua:tell**. The data for each **joshua:tell** of a predication is indexed by the predicate of that predication. To illustrate: in the hardware trouble-shooting example from the Jericho system, all tells of predications of the form `[has-status ...]` will be indexed under the predicate `has-status`.

Tell Metering collects two types of data: *counts* and *times*.

Counts simply keep track of the number of times an event occurs while telling a predication of a particular predicate. Metering collects four counts for each predicate:

1. Tells: How many times a predication of this predicate is told to the database.
2. Matches: How many attempted matches are caused by telling a predication of this predicate. These are matches against forward rule triggers in the forward rule Rete net.
3. Merges: The number of attempted merges in the Rete net occurring while telling predications of this predicate.
4. Rules: How many forward rule firings occurred while telling predications of this predicate.

The *times* collected by Joshua metering tell you how much time is spent in each of the five protocol steps called directly or indirectly by **joshua:tell**. The time reported for a protocol step includes the time in the protocol function and the time in all functions that it calls. It does not include the time spent in other protocol steps. This is referred to as exclusive time of the protocol step.

Tell Metering collects exclusive times for the following protocol steps:

1. **joshua:tell**
2. **joshua:insert**
3. **joshua:justify**
4. **joshua:notice-truth-value-change**
5. **joshua:act-on-truth-value-change**
6. **joshua:map-over-forward-rule-triggers**

In addition exclusive times are collected for two important operations which are not part of the protocol:

1. Matching: The time spent doing the actual matching of told predications against forward rule triggers.
2. Merging: The time spent trying to merge the binding environments generated by the matches.

Exclusive times are collected as histograms. See the section "Expanding Metering Data" in *Program Development Utilities*.Times include paging time and, unless the metering run is done using the without-interrupts option, other process time as well. See the section "Interpreting the Results of a Metering Run" in *Program Development Utilities*.

Here's an example of what a Joshua Tell Metering run might look like. Notice that we can use use the Metering Interface to show us only the counts or times that interest us.

Meter Form (ht:diagnose-circuit nil) **Joshua Tell** Everything

Metering Interface

Metering (6/06/88 23:08:10) JOSHUA-TELL (*TEST*) (Only when Enabled)
Metering (6/07/88 13:44:07) JOSHUA-TELL (*DIAGNOSE-CIRCUIT NIL*)
Metering (6/07/88 13:45:06) JOSHUA-TELL (*DIAGNOSE-CIRCUIT NIL*)
History of Metering Runs

Meter Form | Meter In Process | Re-Meter | Set Display Options | Show Metering Run | Help

Meter Form (ht:diagnose-circuit nil) Joshua Tell Everything

Tell Count	Tell Time Total	Tell Time Avg	Insert Time Total	Insert Time Avg	Map Over Time Total	Map Over Time Avg	Match Count Total	Predicate
1	521	521.00	0	0.00	0	0.00	0	AND-INTERNAL
34	9411	276.79	34791	1023.26	49024	2723.56	34	HAS-OBSERVED-VALUE
34	26963	793.03	38292	1126.24	40040	1820.00	34	HAS-SIMULATED-VALUE
11	2184	198.55	9159	832.64	3806	634.33	6	HAS-STATUS
12	2254	187.83	20611	1717.58	9119	759.92	12	IS-CONNECTED-TO
6	1132	188.67	7132	1188.67	3919	653.17	6	IS-OF-TYPE
22	8218	373.55	30729	1396.77	7955	361.59	0	PORT-DIRECTION

Metering Results

With Joshua Tell Metering you can locate the predicates your program uses the most. You can also find the predicates that spend the most time in **joshua:insert** (or **joshua:justify** or **joshua:map-over-forward-rule-triggers** ...) This information can help you determine how to speed up your program. For example if your program spends a lot of time in **joshua:insert** for a particular predicate, that predicate might be a good candidate for data modeling. See the section "Customizing the Data Index", page 81. If your program spends a great deal of time in **joshua:map-over-forward-rule-triggers** you might try trigger modeling to improve the performance. See the section "Customizing the Matchers Generated by the Rule Compiler", page 102.

Modeling is only advantageous when you use a data structure that is more efficient than the default data structures. Metering can help you choose efficient data structures for your model. You can meter your program before modeling and carefully look at the times for the relevant protocol steps, for example **joshua:insert**. With these numbers in hand, you can implement alternative models; re-meter and compare the numbers. You will immediately be able to see if your modeling was successful.

Here's a simple example of using Joshua metering to measure the performance improvement caused by modeling a predicate.

Tell Count	Insert Time Total	Insert Time Avg	Predicate
1	0	0.00	AND-INTERNAL
4	3370	842.50	GOOD-TO-EAT

Figure 27. Tell metering of the unmodelled good-to-eat predicate.

6.1.2. Joshua Ask Metering

Joshua Ask Metering collects information about each use of **joshua:ask** in your Joshua program. It provides the data in a similar format to that used by Tell Metering, only it shows the protocol steps called while retrieving data from the database and running backward rules.

Tell Count	Insert Time Total	Avg	Predicate
1	0	0.00	AND-INTERNAL
4	274	68.50	GOOD-TO-EAT

Figure 28. Tell metering of the modelled good-to-eat predicate.

Two counts are collected for each predicate:

1. Asks: The number of asks performed for predications of the predicate.

2. Rules: The number of backward rules fired while asking predications of the predicate.

Exclusive time is collected for the following protocol steps:

1. **joshua:ask**
2. **joshua:ask-data**
3. **joshua:fetch**
4. **joshua:ask-rules**
5. **joshua:map-over-backward-rule-triggers**

Note that the time spent in **joshua:ask-questions** is not explicitly reported, but its time is excluded from the other protocol steps.

6.1.3. Joshua Merge Metering

Joshua Merge Metering collects information about the matches and merges that happen in the rete network. See the section "Forward Rule Triggers: the Rete Network", page 27. The data is indexed by the node in the rete network and displayed as several trees. The root of each tree is a rule and the children are the nodes in the network that lead to the triggering of that rule.

In the rete net a single match or merge node can be used in triggering more than one rule. In the Metering Interface these shared nodes are duplicated, so that each rule has its own independent tree of nodes. There is however a visual indicator (an asterisk next to the node name) in the display of each shared node.

The node trees are very similar to the call trees used in other metering types. Initially only the roots of all the trees are visible. There are commands which allow you to show and hide nodes or show and hide node children. This set of commands parallels the commands on call trees. See the section "Exploring a Call Tree" in *Program Development Utilities*. The common commands are invoked with the same gesture for both kinds of trees. See the section "Using the Mouse in the Metering Interface" in *Program Development Utilities*.

Example:

`Meter Form (ht:diagnose-circuit nil)` **Joshua Merge** `Everything`

```
Rule Count   Incl Merge Count   Incl Match Count   Rule/Clause
  Total       Total  Success/    Total  Success/
     4          28               14                 1 | → Rule: HT:ADDER-BACKWARD-1
     4          28               14                 1 | → Rule: HT:ADDER-BACKWARD-2
     2          38               20                 1 | → Rule: HT:ADDER-FORWARD
     2           0                2                 1 | → Rule: HT:ADDER-PORT-DIRECTIONS
     3          54               21                 1 | → Rule: HT:MULTIPLIER-FORWARD
     0          18                9                 1 | → Rule: HT:MULTIPLIER-FORWARD-0-AT-A
     0          18                9                 1 | → Rule: HT:MULTIPLIER-FORWARD-0-AT-B
     3           0                3                 1 | → Rule: HT:MULTIPLIER-PORT-DIRECTIONS
    10         216               30                 1 | ↓ Rule: HT:OBSERVED-WIRE-EQUALITY-1
     -         216               30                 2 | | ↓ Merge:
     -           -               18                 3 | | | ▪ *Match: [HT:HAS-OBSERVED-VALUE ≡OBJECT2 ≡PORT2 ≡VALUE]
     -           -               12                 3 | | | ▪ *Match: [HT:IS-CONNECTED-TO ≡OBJECT1 ≡PORT1 ≡OBJECT2 ≡PORT2]
    10         216               30                 1 | → Rule: HT:OBSERVED-WIRE-EQUALITY-BACKWARDS-1
     1           0                1                 1 | → Rule: HT:POLYBOX-PORT-DIRECTIONS
    12         264               34                 1 | → Rule: HT:WIRE-EQUALITY-1
    12         264               34                 1 | → Rule: HT:WIRE-EQUALITY-BACKWARDS-1
```

Joshua Merge metering helps you locate two different types of wasted work:

1. Rules that may be firing too many times.

2. Rule patterns that may be badly ordered or too general, causing low success percentages for merges.

Probably the most important number provided by Joshua Merge Metering is the merge success percentage. This number gives you the ratio of successful merges to attempted merges. Success percentages for merges can often be improved by re-ordering the clauses in the rule pattern and recompiling the rule. The optimal order depends upon several factors. The two major ones are the number of predications that match this clause and how many variables this clause shares with other clauses.

If you are using trigger modeling, the match success percentages will also be useful. The contract of **joshua:map-over-forward-rule-triggers** is to map over all match nodes that *possibly* match the given predication. The match node itself then determines if it actually matches the predication. If a **joshua:map-over-forward-rule-triggers** method is not selective enough there will be a low match success percentage. Improving the match success rate will decrease the match time (available in Tell Metering) and improve the performance of your Joshua program.

6.2. Choosing Joshua Metering Types

Which type of Joshua metering you should use depends on what your program is doing. Joshua Ask Metering is most useful in programs that include backward chaining. It is also useful for programs that don't do any backward chaining but still use **joshua:ask** frequently to query the database. Joshua Tell Metering is most useful in programs that include forward chaining or for measuring the time spent in setting up an initial database. It is rarely useful for programs that do strictly backward chaining. You should only use Joshua Merge Metering for programs that execute forward rules.

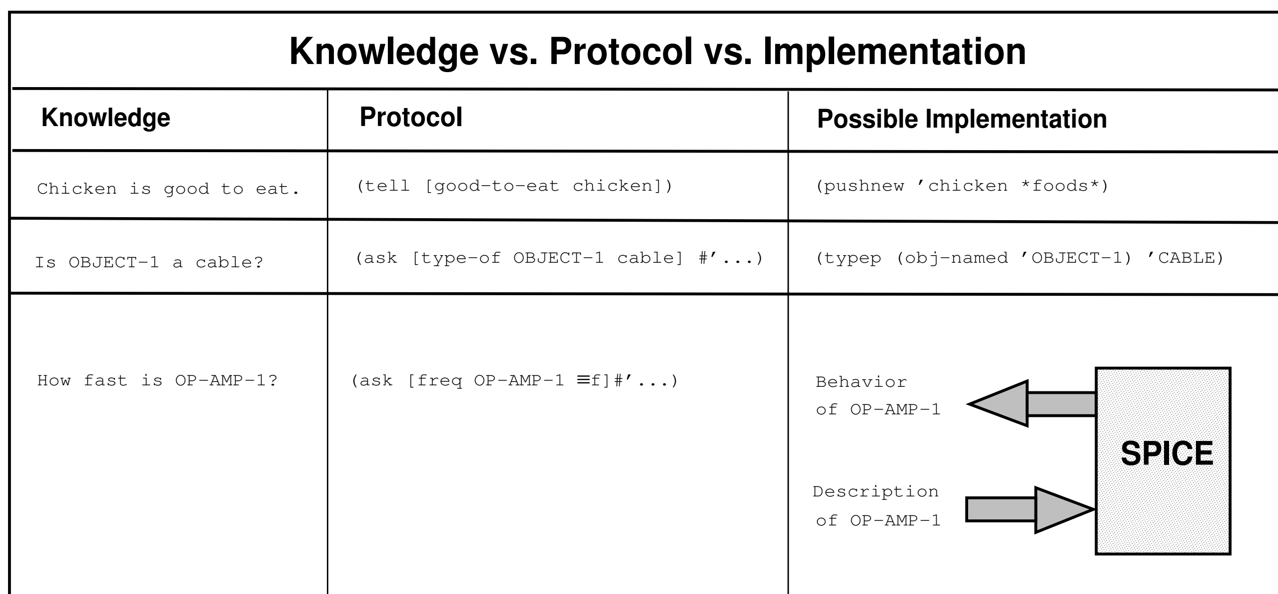
Other system metering types can be useful if a Joshua program includes extensive use of Lisp code or if you want to understand the internal details of how Joshua works.

7. Controlling Data and Rule Indexing

This chapter shows you some basic ways of using the Joshua Protocol to customize Joshua components.

Joshua is a system with "replaceable" parts; that is, you can easily redefine the behavior of any system component. The protocol also makes it straightforward to build an interface that incorporates an existing tool into Joshua without modifying that tool.

In Joshua both the knowledge structures and the contracts of the protocol functions are *distinct from their implementation*. Because the Joshua protocol provides a standard interface for the knowledge structures to communicate with diverse implementations, you can, for example, select any option for data storage without affecting the rule structure or the way that statements involving predications are used for inferencing purposes. Figure29 shows sample interactions between some knowledge structures and their implementations by way of the protocol. As you can see from these examples, the various implementations shown have no effect on the appearance of the knowledge structures, or on the behavior of the top level **joshua:tell** and **joshua:ask** protocol functions as seen by the user. This is because the contract of these functions remains unchanged, regardless of how you implement their behavior.



Knowledge vs. Protocol vs. Implementation

Knowledge	Protocol	Possible Implementation
Chicken is good to eat.	(tell [good-to-eat chicken])	(pushnew 'chicken *foods*)
Is OBJECT-1 a cable?	(ask [type-of OBJECT-1 cable] #'...)	(typep (obj-named 'OBJECT-1) 'CABLE)
How fast is OP-AMP-1?	(ask [freq OP-AMP-1 ≡f]#'...)	Behavior of OP-AMP-1 ← SPICE; Description of OP-AMP-1 → SPICE

Figure 29. Knowledge Structures Can Be Diversely Implemented

Why should you use the protocol to provide implementations other than the defaults provided with Joshua? There is often a gain in efficiency when you use specialized representations for certain key parts of your application. For example,

choosing the right data structure for a problem improves performance. In the case of a system that spends more than half of its time asking questions like, "Is this a cable?" or "Is this a light?", you could improve performance by using a retrieval mechanism that uses **joshua::typep** to answer such questions, as shown in figure 29.

The protocol also allows your application access to the facilities of a specialized tool. For example, as figure29 shows, a Joshua application that needs to answer questions about the behavior of circuits might want to take advantage of a program like SPICE that simulates electronic circuits. Joshua provides a "seamless" way to access customized or external facilities using generic functions like **joshua:ask** and **joshua:tell**.

Or you might want to use a TMS. You get a TMS by creating a mixin that implements the TMS protocol methods (typically this means **joshua:justify**, **joshua:unjustify**, and **joshua:destructure-justification**). With the protocol you can create as many kinds of TMS facilities as your application needs.

How does one customize the protocol? Basically, you need to define a component flavor (using **joshua:define-predicate-model**) and then write methods to implement those portions of the protocol whose behavior you are customizing. This component is then mixed into those predicates (using **joshua:define-predicate**) which need to take advantage of the customized behavior.

Data indexing, rule compilation, rule triggering and truth maintenance are all parts of the protocol, as are their component steps. Since the protocol identifies and makes accessible each step of processing a statement, you can focus modifications very precisely. Typically you would customize the fine grain steps thus preserving the gross structure. Most predicates continue to use most of the default methods. So for example, if you want to change the way data is justified, you define a method for the **joshua:justify** component of **joshua:tell**, but you need not define methods for the data-indexing and rule-locating components of **joshua:tell**. Obviously while customizing any level of the protocol, you need to be aware of functions that work together, (such as **joshua:insert**, **joshua:uninsert**, **joshua:fetch**, and **joshua:clear**), since redefining one of these functions requires redefining the others. But typically the effects of customizing are localized, and changing one area does not require massive changes in the rest of the system.

When should you customize the protocol? Not every application needs customization. Often Joshua's general-purpose facilities serve quite well. However, as you gain more knowledge about your program, you can identify what, if any, portions of it — data, rules, compilation, or truth maintenance — would benefit by customizing, or whether you want to interface to an external tool such as a database. You can thus enrich Joshua's built-in facilities incrementally. In sum, decisions about when and what to customize are entirely application dependent. Joshua's metering tools can be very helpful here, both to identify bottlenecks, and to determine the effect of customizing on your application. See the section "Joshua Metering", page 73.

Here is the list of predicate customizing facilities. We illustrate their use in the examples that follow.

joshua:define-predicate-model
This is the form for specifying new component flavors that be combined and mixed into predicates. Predicate models are not instantiable flavors, they can only be mixed into the definitions of predicates.

joshua:define-predicate
This is the standard predicate-defining form that lets you specify the component flavors each predicate is built on. Predicates are the flavors that can be instantiated; the instances of these are predications (or statements).

joshua:define-predicate-method
This form lets you define methods of component flavors (defined by **joshua:define-predicate-model**) or of predicates (defined by **joshua:define-predicate**) for any of the protocol functions that need redefinition. To undo, use m-X Kill Definition from your Zmacs editor.

joshua:undefine-predicate-model
This form lets you expunge a predicate model definition.

joshua:undefine-predicate
This form lets you expunge a predicate definition.

This chapter discusses and illustrates the following types of customizations:

- Customizing the techniques for Data Indexing
- Customizing the techniques for Rule Indexing
- Customizing the Rule Compiler

The discussion assumes an understanding of the default operation of all the above Joshua features, as described in earlier chapters of this manual.

7.1. Customizing the Data Index

Quick Reference: For a description of the default implementation, see the section "Storing and Retrieving Knowledge in Joshua: the Virtual Database", page 7.

There are many ways in which you can use the Joshua Protocol to change the behavior of your programs; among the most useful is to change the way predications are stored in the virtual database.

Customization of Joshua's data indexing techniques is usually done by defining new methods for **joshua:insert**, **joshua:uninsert**, **joshua:fetch**, and **joshua:clear**. (Under some circumstances, it is preferable to define new methods for **joshua:tell**, **joshua:untell**, and **joshua:ask-data** rather than **joshua:insert**, **joshua:uninsert**, and **joshua:fetch**. One such case is discussed in an example. See the section "Cus-

tomizing the Data Index Without Storing Predications", page 85.) You have to provide a consistent implementation of these methods so that **joshua:tell** knows where to put data, **joshua:untell** knows how to remove specific items, **joshua:ask** knows where to find data, and **joshua:clear** knows how to flush all of it.

This idea is best explained by an example. Suppose you are writing an expert system to mimic the behavior of some animal. When this animal gets hungry, it starts looking for food. Thus, it must have some means of recognizing objects in its world that are good to eat. To start with, you might implement this by defining a predicate like `good-to-eat`, along with a companion **joshua:say** method for a better-looking display:

```
(define-predicate good-to-eat (food-name))

(define-predicate-method (say good-to-eat) (&optional (stream *standard-output*))
  (with-statement-destructured (food-name) self
    (format stream "~&I like to eat ~S." food-name)))
```

Then you can **joshua:tell** your program about some acceptable classes of food. By default, this information is stored in the discrimination net.

```
(tell [and [good-to-eat suan-la-chow-show]
           [good-to-eat kung-pao-chi-ding]
           [good-to-eat ta-chien-chi-ding]
           [good-to-eat lychee-nuts]])
```

Having done this, you can **joshua:ask** your program what's good to eat:

```
(ask [good-to-eat ?] #'say-query :do-backward-rules nil)
I like to eat LYCHEE-NUTS.
I like to eat TA-CHIEN-CHI-DING.
I like to eat KUNG-PAO-CHI-DING.
I like to eat SUAN-LA-CHOW-SHOW.
```

When such a program gets hungry, it searches for a Chinese restaurant.

Eventually you notice that your program spends a great deal of time deciding what to eat. You do a bit of metering and you find that the program is slow because it is fetching predications of the form `[good-to-eat ?food]` (where `?food` may or may not be instantiated), from a rather large discrimination net database, and this is causing too much paging. (The discrimination net potentially takes a page fault on each discrimination net node, the tables in the node, and the list containing the successor nodes. Variables in either the query or the database aggravate this, since more arcs of the discrimination net are followed.)

The solution is to create a data index for `good-to-eat` that stores data about foods in some special place, more easily accessed than the discrimination net. For example, you might just want to have a special variable called `*known-foods*` that is a list of all the foods the program knows about. You'd like **joshua:tell** to push new elements onto that list; depending on whether or not the argument to `good-to-eat` is instantiated, you'd like **joshua:ask** to either search for a particular food, or to loop over the list of known foods. **joshua:clear** should just **joshua::setq** the variable to **joshua::nil**.

Note that often when customizing the data index you give up some generality in return for some performance, or for access to a data source controlled by some other program, such as an external database stored on, for example, a departmental data-processing computer. That's a good trade-off if you're not using the full generality anyway. In this example, we note that food names are always symbols, so we needn't check for strings and such, and can use **joshua::eql** to compare them. We take advantage of the fact that there is only one argument to the `good-to-eat` predicate.

Furthermore, we never **joshua:tell** things like `[good-to-eat ?everything]`, that is, we can forbid the use of variables in the database. These restrictions simplify the implementation considerably, to the point where we need only use a list of food-names and their associated database predications. (One could use a hash table, also, if the number of foods were to get large. Joshua lets you use the whole world of Lisp machine data structures — hash tables, lists, arrays, heaps, or whatever.)

Here is some code that sets up this data index. First, we define the global variable that will be the "database" for known foods.

```
(defvar *known-foods* nil "What's on the menu.")
```

Then we define a predicate model, `good-to-eat-data-model`, for foods the program knows about. Since the methods will be referring to the argument of the predicate frequently, we make the argument an instance variable by putting a **:required-instance-variables** in the **joshua:define-predicate-model** form (as well as a **:destructure-into-instance-variables** in the later **joshua:define-predicate**).

```
(define-predicate-model good-to-eat-data-model () ()
  (:required-instance-variables food))
```

Here is code which implements the strategy we have discussed: using the `variable *known-foods*` as an association list that holds the data and the predication, the **joshua:insert**, **joshua:uninsert**, and **joshua:fetch** methods operate using Lisp list-handling functions.

```
(define-predicate-method (insert good-to-eat-data-model) ()
  ;; tell something about food
  (when (typep food 'unbound-logic-variable)
    (error "You can't possibly mean that everything is good to eat: ~S" self))
  (let ((entry (assoc food *known-foods*)))
    (if entry
        ;; this thing is already known to be good to eat.
        (values (cdr entry) nil)
        ;; not already known -- build a new entry
        (let ((database-predication (copy-object-if-necessary self)))
           ;; this is a new one, put it on the list
           (push (cons food database-predication) *known-foods*)
           (values database-predication t)))))
```

```
(define-predicate-method (uninsert good-to-eat-data-model) ()
  ;;remove food entry from the list
  (setq *known-foods* (delete food *known-foods* :key #'car)))

(define-predicate-method (fetch good-to-eat-data-model) (continuation)
  ;; retrieve some data about known foods
  (typecase food
    (unbound-logic-variable
      ;; wants to succeed once for each possible food
      (loop for (known-food . predication) in *known-foods*
            doing (funcall continuation predication)))
    (otherwise
      ;; wants to know if something in particular is good to eat
      (let ((entry (assoc food *known-foods*)))
        (when entry
              (funcall continuation (cdr entry)))))))

(define-predicate-method (clear good-to-eat-data-model) (clear-database ignore)
  ;; flush all the data about known foods
  (when clear-database
    (setq *known-foods* nil)))


(define-predicate good-to-eat (food)
                      (good-to-eat-data-model default-predicate-model)
  :destructure-into-instance-variables)
```

The only tricky parts are **joshua:insert** and **joshua:fetch**. **joshua:insert** first checks to see that the food argument is bound. That was one of our simplifying assumptions. Then it looks to see if it already has a database entry for that food. If so, the database predication is returned along with the **joshua::nil** indicating that the entry was already in the database. If not, the predication is copied if necessary. This will ensure that the predication put in the database is not ephemeral in any way. The database predication is returned, along with **joshua::t** to indicate that the database predication is newly inserted in the database.

The method for **joshua:fetch** deserves some explanation. The **joshua::unbound-logic-variable** check is to distinguish between `(ask [good-to-eat suan-la-chow-show] ...)` and `(ask [good-to-eat ?x] ...)`, that is, whether or not the query has a logic variable in its argument. The former asks a question like "Is this particular thing good to eat?", while the latter says "For everything you can prove is good to eat, do ...". So **joshua:fetch** must check to see what sort of question is being asked. If the query has a variable food, we must succeed once with each good-to-eat predication in the database. If the query is about a specific food, we look it up in the database and succeed if it is there.

Here's a hypothetical interaction with the program:

```
(clear)

*known-foods* → NIL

(tell [and [good-to-eat suan-la-chow-show]
           [good-to-eat kung-pao-chi-ding]
           [good-to-eat ta-chien-chi-ding]
           [good-to-eat lychee-nuts]])

*known-foods* → ((LYCHEE-NUTS . [GOOD-TO-EAT LYCHEE-NUTS])
                 (TA-CHIEN-CHI-DING . [GOOD-TO-EAT TA-CHIEN-CHI-DING])
                 (KUNG-PAO-CHI-DING . [GOOD-TO-EAT KUNG-PAO-CHI-DING])
                 (SUAN-LA-CHOW-SHOW . [GOOD-TO-EAT SUAN-LA-CHOW-SHOW]))

(ask [good-to-eat ?] #'say-query :do-backward-rules nil) →
I like to eat LYCHEE-NUTS.
I like to eat TA-CHIEN-CHI-DING.
I like to eat KUNG-PAO-CHI-DING.
I like to eat SUAN-LA-CHOW-SHOW.

(clear)

*known-foods* → NIL
```

So by giving up some generality, we can gain some performance. It is not hard to use data structures other than association lists. Just replace the list insertion and lookup functions with ones appropriate to your data structure.

7.1.1. Customizing the Data Index Without Storing Predications

In some cases, you won't want to store entire predication objects in the database. For example, you might be using some external database for some predicates. Joshua provides a different level of the data protocol for customizing such applications. For such applications, you would provide **joshua:tell**, **joshua:untell**, **joshua:ask-data**, and **joshua:clear** methods. Note that some types of predicates require that information be stored in the database predication. In particular, the forward-chaining rule mechanism stores state information in database predications, and the system-supplied TMS mixin, **ltms:ltms-mixin**, stores its support information in database predications. So a requirement for using this type of customization on predicates is that the predicates be used only in backward chaining. (However, it is sometimes possible to use the **joshua:expand-forward-rule-trigger** protocol function to look for the data during forward rule triggering. See the dictionary entry for this protocol function for examples.)

For an example, we turn to the `good-to-eat` example introduced in another section. (See the section "Customizing the Data Index", page 81.)

We'll make the same assumptions that were made in that example — that the foods in the database are always symbols, never variables. For simplicity we'll impose a further restriction — that all assertions must be stated positively. This model will not allow `(tell [not [good-to-eat yu-shiang-giant-beetle]])`.

The **joshua:insert**, **joshua:uninsert**, and **joshua:fetch** methods of that previous example dealt with database predications. So for this type of customization, we'll move to the next higher level of the protocol — **joshua:tell**, **joshua:untell**, and **joshua:ask-data**. Another way to think about why we have to move up a level is to notice that **joshua:tell** and **joshua:untell** will try to **joshua:justify** and **joshua:unjustify**, which need database predications.

Here is the new code:

```
(define-predicate-method (tell good-to-eat-data-model)
                         (truth-value justification)
  (declare (ignore justification))
  ;; tell something about food
  (unless (eql truth-value *true*)
    (error "Only positive assertions are allowed for ~S" self))
  (when (typep food 'unbound-logic-variable)
    (error "You can't possibly mean that everything is good to eat: ~S" self))
  (pushnew food *known-foods*))


(define-predicate-method (untell good-to-eat-data-model) ()
  ;; remove food name from the list
  (setq *known-foods* (delete food *known-foods*)))

(define-predicate-method (ask-data good-to-eat-data-model)
                         (truth-value continuation)
  ;; given the contents of *known-foods*, unify against the query
  ;; and call the continuation
  (unless (eql truth-value *true*)
    (signal 'ji:model-can-only-handle-positive-queries
            :query self
            :model 'good-to-eat-data-model))
  (typecase food
    (unbound-logic-variable
      ;; wants to succeed once for each possible food
      (loop for database-food in *known-foods*
            doing (with-unification
                    ;; bind the variable in the query and go on
                    (unify food database-food)
                    (stack-let ((support
                                  `(,self ,truth-value good-to-eat-data-model)))
                      (funcall continuation support)))))
    (otherwise
      ;; wants to know if something in particular is good to eat
      (when (member food *known-foods*)
        (stack-let ((support `(,self ,truth-value good-to-eat-data-model)))
          (funcall continuation support))))))
```

```
(define-predicate-method (clear good-to-eat-data-model)
                         (clear-database ignore)
  ;; flush all the data about known foods
  (when clear-database
    (setq *known-foods* nil)))

(define-predicate good-to-eat (food)
                  (good-to-eat-data-model default-predicate-model)
  :destructure-into-instance-variables)
```

The **joshua:tell** method is very straightforward — after checking our assumptions, it makes sure the food is on the list. The **joshua:untell** and **joshua:clear** methods are almost identical to our previous example.

The **joshua:ask-data** method has to do the same positiveness check that **joshua:tell** did, but **joshua:ask-data** should signal the **ji:model-can-only-handle-positive-queries** condition if the query isn't positive. **joshua:ask-data** does the same **joshua::unbound-logic-variable** test that **joshua:fetch** in the previous example did. When the argument is unbound, **joshua:ask-data** must bind it before calling the continuation. In this case, the derivation provided by the **joshua:ask-data** method is just the symbol `good-to-eat-data-model`, to tell programs what flavor provided this answer.

From the point of view of the user of **joshua:tell** and **joshua:ask**, the behavior of the customized system is just the same as before, but this version of the program does not touch the discrimination net. The information that used to go into the discrimination net is now stored in a different place, namely the value of `*known-foods*`. Note in the following examples that **joshua:tell** and **joshua:ask** act exactly as before, but the value of `*known-foods*` changes.

```
(clear)

*known-foods* → NIL

(tell [and [good-to-eat suan-la-chow-show]
           [good-to-eat kung-pao-chi-ding]
           [good-to-eat ta-chien-chi-ding]
           [good-to-eat lychee-nuts]])

*known-foods* → (LYCHEE-NUTS TA-CHIEN-CHI-DING KUNG-PAO-CHI-DING SUAN-LA-CHOW-SHOW)

(ask [good-to-eat ?] #'say-query :do-backward-rules nil) →
I like to eat LYCHEE-NUTS.
I like to eat TA-CHIEN-CHI-DING.
I like to eat KUNG-PAO-CHI-DING.
I like to eat SUAN-LA-CHOW-SHOW.
```

In this case as before, other data structures could easily be substituted for the list.

If the data structure were made just slightly more complicated, negative assertions could be handled. **joshua:tell** just needs to remember the current truth value, and **joshua:ask-data** needs to succeed only on foods which have the same database truth value as the query.

7.2. Customizing the Rule Index

Quick Reference: For a description of the default rule-indexing implementation, see the section "The Joshua Rule Facilities ", page 23. This section disusses the process of changing the way the system stores, removes, and looks up rule triggers.

A trigger is an object that can be used to invoke a particular rule or question. Exactly what the trigger object is depends on the rule type. For a forward rule, the triggers are Rete match nodes, and "invoke" means a call to some Rete network code to start the match process. For a backward rule, the (unique) trigger is a function that does both the matching and the rule-body execution. For questions, a trigger is just the name of a function to call; if called with reasonable arguments, it will ask its question. The default implementation stores trigger objects in a discrimination net.

The impetus for customizing rule indexing is that many systems spend much of their time looking for applicable rules and questions, as opposed to executing them. For example, if you **joshua:tell** the system a fact, in many cases most of the resulting runtime is spent looking for forward rules, rather than executing them. If this is the case, a customzied rule-trigger index can help.

There are four operations you can do with rule triggers, namely, adding new triggers, deleting existing triggers, locating the place which a trigger should be stored in (or removed from), iterating over triggers to find rules or questions that can execute. If you provide a consistent alternative implementation of the protocol functions that do these operations, you've changed the way your program looks for rules, and perhaps achieved improved performance.

Let's consider our food example again, namely the `good-to-eat` example used in the section on customized data indexing. see the section "Customizing the Data Index", page 81.

In brief, this example identifies foods that are good to eat, so that the program's subject knows what foods to look for when it gets hungry. There might be a number of backward-chaining rules in the system that have triggers resembling `[good-to-eat ?x]` or `[good-to-eat suan-la-chow-show]`. The problem is that the default way of storing triggers goes through a very general discrimination net, whereas in this case, there is only one interesting datum, namely the argument to `good-to-eat`. We want to produce a system that, when looking for backward rules to solve `good-to-eat` goals, will discriminate only on that argument.

Here's an example of such an implementation for backward rules (to keep things simple). First, we define a structure in which to store rule triggers. Since the trigger pattern contains either a logic variable or a symbol, a hash table for symbols

and a separate list for the variable case will do. That is, we have one place to store triggers like [good-to-eat suan-la-chow-show], and another for things like [good-to-eat ?x].

```
(defvar *good-to-eat-constant-triggers* (make-hash-table :size 10))
(defvar *good-to-eat-variable-triggers* nil)
```

Next, we want **joshua:add-backward-rule-trigger**, **joshua:delete-backward-rule-trigger**, **joshua:locate-backward-rule-trigger**, and **joshua:map-over-backward-rule-triggers** to use these data structures instead of the defaults.

We need not redefine **joshua:add-backward-rule-trigger** and **joshua:delete-backward-rule-trigger**, because both use a storage location provided by **joshua:locate-backward-rule-trigger**. So it is sufficient to define a method for **joshua:locate-backward-rule-trigger** and **joshua:map-over-backward-rule-triggers**.

We begin by defining our good-to-eat predicate model.

```
(define-predicate-model good-to-eat-trigger-model () ()
  (:required-instance-variables food))
```

Next we define the **joshua:locate-backward-rule-trigger** method. The code for this considers two cases.

- If the predication passed to **joshua:locate-backward-rule-trigger** contains a logic variable, the method calls the continuation on the *good-to-eat-variable-triggers* list. If the continuation changed the list of triggers, the method updates it.

- If the predication passed to **joshua:locate-backward-rule-trigger** contains a constant, the method calls the continuation on the appropriate elements of the *good-to-eat-constant-triggers* table. If the continuation changed the list of triggers, the method updates the table.

```
(define-predicate-method
  (locate-backward-rule-trigger good-to-eat-backward-trigger-model)
  (truth-value continuation &optional ignore)
  ;; call continuation on list of triggers for backward rules that
  ;; solve good-to-eat goals
  (typecase food
    (unbound-logic-variable
      ;; the argument is an unbound logic variable
      (multiple-value-bind (new-triggers triggers-changed-p canonical-trigger)
          (funcall continuation *good-to-eat-backward-variable-triggers*)
        (when triggers-changed-p
          (setq *good-to-eat-backward-variable-triggers* new-triggers))
        canonical-trigger))
    (otherwise
      ;; the argument is a constant
      (let ((list-of-triggers
              (gethash food *good-to-eat-backward-constant-triggers*)))
        (multiple-value-bind (new-triggers triggers-changed-p canonical-trigger)
            (funcall continuation list-of-triggers)
          (when triggers-changed-p
            (if (null new-triggers)
                ;; if they're all undefined, nuke this entry
                (remhash food *good-to-eat-backward-constant-triggers*)
                (setf (gethash food *good-to-eat-backward-constant-triggers*)
                      new-triggers)))
          canonical-trigger)))))
```

joshua:map-over-backward-rule-triggers must apply the continuation to all applicable backward rule triggers that solve `good-to-eat` goals. Our method first maps over the variable trigger list, since all variable triggers must be probed regardless of whether the goal has a logic variable or a constant in it.

If the goal has a logic variable in it (for example `(ask [good-to-eat ?what]... ))`, the method must also apply the continuation to all the variable triggers, since any constant matches the logic variable in the goal.

If the goal has a constant in it (for example `(ask [good-to-eat honey] ...))`, **joshua:map-over-backward-rule-triggers** only needs to call the continuation on trigger objects with matching constants.

```
(define-predicate-method
  (map-over-backward-rule-triggers good-to-eat-backward-trigger-model) (continuation)
  ;; map continuation over triggers of backward rules that
  ;; solve good-to-eat goals
  ;; first do all variable triggers
  (mapc continuation *good-to-eat-backward-variable-triggers*)
  (typecase food
    (unbound-logic-variable
      ;; food is unbound, so have to map over all remaining triggers
      (maphash #'(lambda (key value)
                   (ignore key)
                   (mapc continuation value))
               *good-to-eat-backward-constant-triggers*))
    (otherwise
      ;; food is bound, so map over just those triggers that will match
      (mapc continuation
            (gethash food *good-to-eat-backward-constant-triggers*)))))
```

The last step is defining a good-to-eat predicate.

```
(define-predicate good-to-eat (food) (good-to-eat-trigger-model
          good-to-eat-data-model default-predicate-model)
  :destructure-into-instance-variables)
```

With these definitions in place, any backward-chaining rules that trigger on good-to-eat will store their triggers in either *good-to-eat-variable-triggers* or *good-to-eat-constant-triggers*. Presumably, searching these data structures is faster than the general-purpose method Joshua provides by default.

Here are some examples. Assume the following (admittedly silly) rule:

```
(defrule not-tofuud (:backward)
  IF (progn (format t "~&Not-Tofuud rule: ?FOOD → ~S" ?food) t)
  THEN [good-to-eat ?food])
```

Before we compile this rule, the list *good-to-eat-variable-triggers* is **joshua::nil**. However, after the rule is compiled, this list is bound to something that looks like:

```
(#S(JI::BACKWARD-TRIGGER :TRUE-ENTRIES ((NOT-TOFUUD NIL
    [GOOD-TO-EAT #<UNBOUND-LOGIC-VARIABLE ?FOOD 51037342>]))
                          :FALSE-ENTRIES NIL))(#<BACKWARD-TRIGGER 14702210>)
```

that is, it is a list of backward rule triggers that trigger on good-to-eat with a logic variable argument.

For rules that trigger on good-to-eat with a constant argument, we use an **joshua::eql** hash table (it assumes the arguments to good-to-eat can be compared with **joshua::eql**). Consider the following (also silly) rule:

```
(defrule tofuud (:backward)
  IF (progn (format t "~&Tofuud rule.") t)
  THEN [good-to-eat tofu])
```

Before compilation of this rule, *good-to-eat-constant-triggers* is an empty table, that prints something like this:

```
#<Table 0/0 51004361>
```

After we compile the rule, however, the table contains one entry, keyed on the symbol tofu:

```
  (describe *good-to-eat-constant-triggers*)
#<Table 1/1 51004361> is a table with 1 entry.
Test function for comparing keys = EQL, hash function = CLI::XEQLHASH
Do you want to see the contents of the hash table? (Y or N) Yes.
 TOFU → (#S(JI::BACKWARD-TRIGGER :TRUE-ENTRIES ((TOFUUD NIL [GOOD-TO-EAT TOFU]))
                                 :FALSE-ENTRIES NIL))
#<Table 1/1 51004361>
```

Consider how the following query manages to operate:

```
(ask [good-to-eat ?what] #'say-query)
Not-Tofuud rule: ?FOOD → #<UNBOUND-LOGIC-VARIABLE ?WHAT 50474332>
#<UNBOUND-LOGIC-VARIABLE ?WHAT 50474332> is good to eat.
Tofuud rule.
TOFU is good to eat.
NIL
```

In processing this query, Joshua avoids the conventional method of looking for rule triggers, and instead looks in the list *good-to-eat-variable-triggers* and probes the hash table *good-to-eat-constant-triggers*.

Proper design of the data structures that your program uses for looking up rules can drastically affect the program's performance. In this example, we noted that good-to-eat had only one argument that was either a symbol or a logic variable. We were able to exploit this restriction to implement a more efficient way to look up rule triggers than could be provided in a general-purpose implementation.

There is a completely analogous group of methods for a forward chaining implementation.

7.3. Customizing the Rule Compiler

Quick Reference: For a description of the default implementation, see the section "The Joshua Rule Compiler", page 26.

The rule compiler can be customized in two ways. First, the trigger patterns of a forward rule and the actions of a backward rule (i.e. the *If-part* of a rule) may be *expanded* into other structures in a process similar to macro expansion. This allows the If-Part of the rule to present a declarative appearance even when it actually takes procedural actions. Secondly, the rule compiler can generate specialized or optimized pattern matchers that take advantage of the trigger indexing techniques used for the patterns.

The first of these kinds of customizations is controlled by the **joshua:expand-forward-rule-trigger** and the **joshua:expand-backward-rule-action** protocol methods, the second type is controlled by the **joshua:write-forward-rule-full-matcher**, **joshua:write-forward-rule-semi-matcher**, **joshua:positions-forward-rule-matcher-can-skip** and the **joshua:write-backward-rule-matcher** protocol methods.

Customizing the Expansion of a Forward Rule

The If-part of a forward-chaining rule is eventually translated into a rete network,See the section "Forward Rule Triggers: the Rete Network", page 27. How this translation is conducted is controlled by the **joshua::expand-forward-rule-triggers** protocol function. This section will explain what this protocol function does and give examples of how its capabilities can be used to gain advanced capabilities in forward-chaining rules.

The Contract of the Generic Function joshua:expand-forward-rule-trigger

joshua:expand-forward-rule-trigger is called once for each predication included in the trigger of the rule. Its job is to return a list structure that explains to the rule compiler how to process the pattern.

For example in the following rule:

```
(defrule foobar (:forward)
  If [and [foo1 ?x ?y] :support ?f1
          [not [foo2 ?y ?z]] :support ?f2
          ]
  Then [foo3 ?x ?y ?z])
```

joshua:expand-forward-rule-trigger will be called three times (once for the entire **joshua::and** and then once for each predication inside the **joshua::and**). **joshua:expand-forward-rule-trigger** takes four arguments: the pattern to expand, the name of its :support variable (or nil), its truth-value and the entire If-part (which can be treated as the "context" of the pattern). Thus, the arguments passed in for these three calls wil be:

```
[and [foo1 ?x ?y] :support ?f1
     [not [foo2 ?y ?z]] :support ?f2] nil *true* and <the whole If-part>
[foo1 ?x ?y] ?f1 and *true* <the whole If-part>
[foo2 ?y ?z] ?f2 *false* <the whole If-part>
```

Note that although we have displayed the patterns as if they were predications, this is not actually true. **joshua:expand-forward-rule-trigger** runs at compile time and manipulates a source-code representation of predications and logic-variables, see the section "The Source Representaton of Predications and Logic-variables".

joshua:expand-forward-rule-trigger should return a list structure (called a *trigger-description*) which must be one of the following forms:

1. (`:match` *pattern name truth-value*). This trigger description informs the rule compiler that the current trigger should be treated simply as a pattern to be matched.

- *pattern* is the predication that represents the pattern to be matched.
- *name* is the logic variable which the rule triggering mechanisms should bind to the predication that matched this trigger.
- *truth-value* (which in the current implementation should be either **joshua:*true*** or **joshua:*false***) is the truth value which the matching predication is required to have in order to trigger the rule.

2. (:and *trigger-descriptions*) This trigger description informs the rule compiler that the current pattern is actually a conjunction of patterns all of which must be matched to trigger the rule. The system-provided default method for AND predications returns this type of trigger description. The second element of the trigger description must be a list of trigger descriptions, i.e. lists returned by calling **joshua:expand-forward-rule-trigger**.

3. (:or *trigger-descriptions*) This trigger description informs the rule compiler that the current pattern is actually a disjunction of patterns any of which must be matched to trigger the rule. The system provided default method for OR predications returns this type of trigger description. The second element of the trigger description must be a list of trigger descriptions, i.e. lists returned by calling **joshua:expand-forward-rule-trigger**.

4. (:procedure *lisp-expression name*) This trigger description informs the rule compiler that the current trigger is not a pattern to be matched, but rather a Lisp expression that appears in the trigger position. Such expressions are executed once all proceeding patterns in the rule have been matched. The expression can act as a filter by returning either **joshua::t** or **joshua::nil**. **joshua::t** indicates success; in this case the bindings accumulated up to this point are considered acceptable and rule triggering continues. **joshua::nil** indicates failure; in this case the bindings are considered unacceptable.

   The expression can also act as a generator in which it produces several new sets of bindings each of which is consistent with the bindings that were in effect when the rule was triggered. To do this it should bind whatever logic-variables it wants to and then call **joshua:succeed**. **joshua:succeed** takes a rest-argument; the rule compiler will arrange for this values passed to **joshua:succeed** to be bound to the logic-variable which is the third element of the trigger description.

   See the function **joshua:succeed**, page 232.

5. (:ignore) This trigger description informs the rule compiler that it should ignore this trigger. The are two reasons for using this type of trigger description. The first is to allow a rule to have patterns included in it simply for the sake of clarity. The second is to include patterns only to specify context.

Using joshua:expand-forward-rule-trigger

A Procedural trigger description can be used to implement a mixed-chaining strategy in which a forward-rule trigger invokes backward chaining capabilities. This would be useful if it is known that a particular type of predication is never actually asserted but is only deduced by backward chaining rules.

The following rule is how one would implement this mixed-chaining strategy if it were known that FOO2 predications are only deduced by backward chaining rules:

```
(define-predicate foo1 (a b))
(define-predicate foo2 (a b))
(define-predicate backward-foo2 (a b))
(define-predicate foo3 (a b c))

(defrule foo (:forward)
  If [and [foo1 ?a ?b]
          (ask [foo2 ?b ?c]
               #'(lambda (ignore) (succeed)))]
  Then [foo3 ?a ?b ?c])

(defrule foo2-backward (:backward)
  If [backward-foo2 ?b ?a]
  Then [foo2 ?a ?b])
```

The structure of the rete network for this rule is a simple linear chain consisting of a match node followed by a procedural node (acting as a generator) as shown in figure 40.

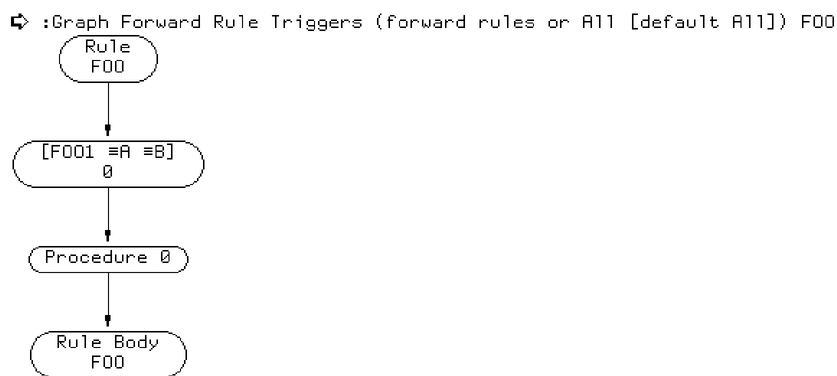


Figure 30. Graph of the Mixed Chaining Rule Foo

If we execute the following two **joshua:tell**'s then the rule will be triggered by the second statement which matches the first pattern of the rule. Execution then proceeds to the procedural node which chains backward using the rule FOO2-BACKWARD. This is shown in figure 41.
However this rule can be made more declarative appearing by using

```
(tell [backward-foo2 3 2])
 ▶ Telling predication [BACKWARD-FOO2 3 2]
[BACKWARD-FOO2 3 2]
T
⇨
(tell [foo1 1 2])
 ▶ Telling predication [FOO1 1 2]
 ▶ Asking predication [FOO2 2 ≡C]
 ▶ Trying backward rule FOO2-BACKWARD  (Goal... )
   ▶ Asking predication [BACKWARD-FOO2 ≡C 2]
 ▶ Succeeding backward rule FOO2-BACKWARD
 ▶ Firing forward rule FOO (1 trigger)
   ▶ Telling predication [FOO3 1 2 3]
```

Figure 31. Trace of The Mixed Chaining Rule Foo

joshua:expand-forward-rule-trigger as follows:

```
(define-predicate-model mixed-chaining-mixin () ())

(define-predicate-method
  (expand-forward-rule-trigger mixed-chaining-mixin)
  (name truth-value ignore)
  (let ((query (if (eql truth-value *true*)
                   self
                   `[not ,self])))
    `(:procedure (prog1 nil
                        (ask ,query
                             #'(lambda (ignore)
                                 (succeed))))
      ,name)))

(define-predicate foo2 (a b)
  (mixed-chaining-mixin default-predicate-model))

(defrule foo (:forward)
  If [and [foo1 ?a ?b]
          [foo2 ?b ?c]]
  Then [foo3 ?a ?b ?c])
(clear)
(tell [backward-foo2 3 2])
(tell [foo1 1 2])
```

Now the rule `FOO` appears to simply match two patterns. However, it actually compiles into exactly the same rete network as shown in figure 30.

A More Advanced Version of Mixed-chaining in joshua:expand-forward-rule-trigger

Sometimes using **joshua:ask** in the trigger part of a rule may not be the appropriate way to achieve a mixed chaining strategy. One reason, is that **joshua:ask** queries the world for facts that are deducible at that moment. If a new fact arrives later that would have made the goal deducible, **joshua:ask** will, of course,

not notice this. However, forward chaining rules should draw conclusions whenever the data warrants the deduction.

A solution to this problem is to use a more explicit form of reasoning in which goal directed reasoning is conducted by forward rules which are triggered by explicit predications stating the existence of a goal.

Here is an alternative mixed chaining scheme which implements backward chaining by explicitly telling `show` predications. These trigger forward rules which then work to find a way to satisfy the goal included in the `show` statement.

For example, the following rule:

```
(defrule foo2-explicit-goal (:forward)
  If [and [show [foo2 ?a ?b]]
          [backward-foo2 ?b ?a]]
  Then [foo2 ?a ?b])
```

Will deduce `FOO2` anytime that `BACKWARD-FOO2` is asserted and there is a `SHOW` predication stating that we want this conclusion to be drawn. The rule is more flexible than a backward rule, since it does not depend on the relative order of posting the goal and asserting the data necessary to deduce it. (Of course, this rule is also less efficient than a backward rule).

We can use **joshua:expand-forward-rule-trigger** just as we did in the previous section to make the rule `FOO` use this form of mixed chaining while retaining its declarative appearance, as follows:

```
(define-predicate-model mix-chain-mixin ()
  ())

(defvar *inside-alternative-backward-chaining-mixin* nil)

(define-predicate-method
  (expand-forward-rule-trigger mix-chain-mixin)
  (name truth-value context)
  (if *inside-alternative-backward-chaining-mixin*
      `(:match ,self ,name ,truth-value)
      (let ((*inside-alternative-backward-chaining-mixin* t))
        (let ((query (if (eql truth-value *true*)
                         self
                         `[not ,self])))
          `(:and
             ,(expand-forward-rule-trigger
                `(tell [show ,query]) nil *true* context)
             ,(expand-forward-rule-trigger
                self name truth-value context))))))

(define-predicate show (predication))

(define-predicate foo2 (a b)
  (mix-chain-mixin default-predicate-model))
```

This **joshua:expand-forward-rule-trigger** method expands the FOO2 pattern of the rule into two components. The first **joshua:tell**'s the SHOW statement that triggers the FOO2-EXPLICIT-GOAL rule. The second is a simple match node that waits for the FOO2 goal to become true. The **joshua:expand-forward-rule-trigger** method is somewhat tricky because it wants to expand the intial [foo2 ...] pattern into two nodes, one of which **joshua:tell**s [show [foo2 ...]] and the other of which matches [foo2 ...]. A special variable is bound to prevent an infinite recursion in the expansion of this pattern.

Figure42shows the Rete net for this rule.

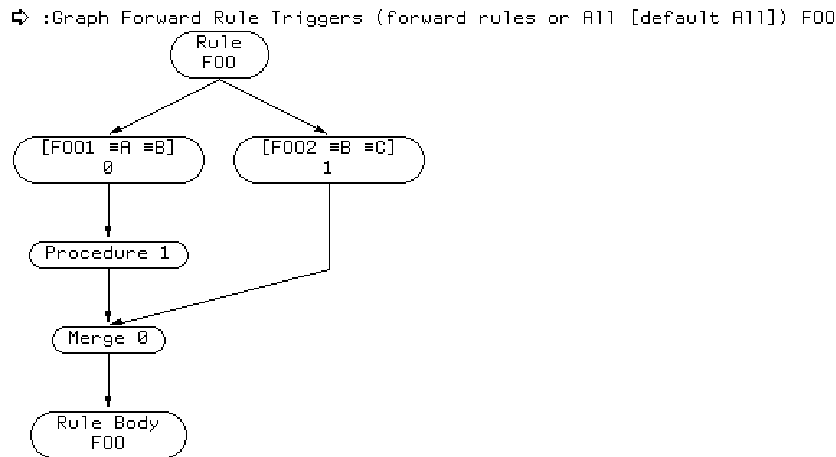


Figure 32. Graph of Mixed Chaining Rule Foo

Notice that the rule contains two match nodes, one for each pattern. The match node for the FOO1 pattern leads to a procedural node which **joshua:tell**s a [show [foo2 ...]] predication and then **joshua:succeed**s. Following this the two paths merge. If the Foo1 statement is asserted first the rule will assert the SHOW statement which will cause the FOO2-EXPLICIT-GOAL rule to wait for a FOO2-BACKWARD statement. At which point the FOO2-EXPLICIT-GOAL rule will assert a FOO2 statement which will match the other trigger pattern of the FOO rule. If the facts are asserted in the other order, the rule will also deduce the desired conclusion, as shown in figures 43 and

Using :ignore in joshua:expand-forward-rule-trigger

Here's an example using the :ignore trigger description:

```
(defrule adder-forward (:forward)
  If [and [type-of ?a adder]
          [Value-of addend ?a ?value-1]
          [Value-of augend ?a ?value-2]]
  Then `[value-of output ?a ,(+ ?value-1 ?value-2)])
```

A trigger-indexing scheme might be used which guarantees that this rule will only be triggered by Value-of assertions that describe the values of the ADDEND and AUGEND of adders. In such a case the first pattern is required during rule compilation

```
(tell [backward-foo2 3 2])
 ▶ Telling predication [BACKWARD-FOO2 3 2]
[BACKWARD-FOO2 3 2]
T
➪
(tell [foo1 1 2])
 ▶ Telling predication [FOO1 1 2]
 ▶ Telling predication [SHOW [FOO2 2 ≡C]]
 ▶ Firing forward rule FOO2-EXPLICIT-GOAL (2 triggers)
   ▶ Telling predication [FOO2 2 3]
 ▶ Firing forward rule FOO (2 triggers)
   ▶ Telling predication [FOO3 1 2 3]
[FOO1 1 2]
```

Figure 33. Trace of Explicitly Controlled Mixed Chaining

```
(tell [foo1 1 2])
 ▶ Telling predication [FOO1 1 2]
 ▶ Telling predication [SHOW [FOO2 2 ≡C]]
[FOO1 1 2]
T
➪
(tell [backward-foo2 3 2])
 ▶ Telling predication [BACKWARD-FOO2 3 2]
 ▶ Firing forward rule FOO2-EXPLICIT-GOAL (2 triggers)
   ▶ Telling predication [FOO2 2 3]
   ▶ Firing forward rule FOO (2 triggers)
     ▶ Telling predication [FOO3 1 2 3]
[BACKWARD-FOO2 3 2]
```

Figure 34. Trace of Explicitly Controlled Mixed Chaining

to inform the **joshua:locate-forward-rule-trigger** method that it is indexing patterns having to do with adders. However, once such a trigger-indexing scheme is established the first pattern is actually redundant.

```
(define-predicate-method
  (expand-forward-rule-trigger type-of-model) (ignore ignore ignore)
  `(:ignore))


(define-predicate type-of (object type)
                 (type-of-model default-protocol-implementation-model))
```

Customizing the Expansion of a Backward Rule

What the Backward Rule-compiler Does to the Actions of a Rule

The backward rule compiler turns the If-part of a rule into a series of nested **joshua:ask**'s. For example, the actions of the following rule:

```
(defrule foobar (:backward)
  If [and [foo1 ?x ?y] :support ?f1 :do-backward-rules nil
          [not [foo2 ?y ?z]] :support ?f2
          ]
  Then [foo3 ?x ?y ?z])
```

are converted into a highly optimized version of the following code:

```
(ask [foo1 ?x ?y]
     #'(lambda (support2196)
         (unify ?f1 support2196)
         (ask [not [foo2 ?y ?z]]
              #'(lambda (support2197)
                  (unify ?f2 #:support2197)
                  (let ((ji::rule-support
                          (list ji::.goal. ji::.truth-value.
                                '(rule foobar)
                                support2196 support2197)))
                    (funcall ji::.continuation. ji::rule-support))))
         :do-backward-rules nil))
```

The backward rule compiler also handles the keyword arguments which can be attached to patterns in the If-part of the rule. See the section "Advanced Features of Joshua Rules", page 24.

The Contract of the Generic Function joshua:expand-backward-rule-action

The **joshua:expand-backward-rule-action** protocol function controls how the conversion is performed.

joshua:expand-backward-rule-action is called once for each predication included in the If-part of the rule. Its job is to return a list structure that explains to the rule compiler how to process the pattern.

For example in the following rule:

```
(defrule foobar (:backward)
  If [and [foo1 ?x ?y] :support ?f1 :do-backward-rules nil
          [not [foo2 ?y ?z]] :support ?f2
          ]
  Then [foo3 ?x ?y ?z])
```

joshua:expand-backward-rule-action will be called three times (once for the entire **joshua::and** and then once for each predication inside the **joshua::and**). **joshua:expand-backward-rule-action** takes five arguments: the pattern to expand, the name of its :support variable (or nil), its truth-value, the value of the keyword arguments attached to this pattern that should be passed onto **joshua:ask** (e.g. :do-backward-rules and :do-questions) and the entire If-part (which can be treated as the "context" of the pattern). Thus, the arguments passed in for these three calls wil be:

```
[and [foo1 ?x ?y] :support ?f1 :do-backward-rules nil
     [not [foo2 ?y ?z]] :support ?f2] nil *true* (t t) <the whole If-part>
[foo1 ?x ?y] ?f1 *true* (nil t) <the whole If part>
[foo2 ?y ?z] ?f2 *false* (t t) <the whole If part>
```

Note that although we have displayed the patterns as if they were predications, this is not actually true. **joshua:expand-backward-rule-action** runs at compile time and manipulates a source-code representation of predications and logic-variables, see the section "The Source Representaton of Predications and Logic-variables".

joshua:expand-backward-rule-action should return a list structure (called a *action-description*) which must be one of the following forms:

1. (`:match` *pattern name truth-value* ask-keyword-args). This action description informs the rule compiler that the current action should be treated simply as a pattern to be **joshua:ask**'ed. This action will compile into an **joshua:ask** form whose continuation will perform the actions following this one.

   - *pattern* is the source representation of the predication that should be **joshua:ask**'ed. This is normally just the first argument to **joshua:expand-backward-rule-action**.

   - *name* is the name of a logic variable which should be bound to the query-support passed by **joshua:ask** to its continuation; this allows procedural code in the If-Part of the rule to examine the support for the various actions.

   - *truth-value* (which in the current implementation should be either **joshua:*true*** or **joshua:*false***) is the truth value which the matching predication is required to have in order to satisfy the **joshua:ask**.

   - The values of the keyword arguments to be passed to **joshua:ask**. This should normally be identical to the equivalent argument passed into **joshua:expand-backward-rule-action**.

2. (`:and` *action-descriptions*) This action description informs the rule compiler that the current pattern is actually a conjunction of actions all of which must be satisfied. The system-provided default method for AND predications returns this type of action description. The second element of the trigger description must be a list of action descriptions, i.e. lists returned by calling **joshua:expand-backward-rule-action**.

3. (`:or` *action-descriptions*) This action description informs the rule compiler that the current pattern is actually a disjunction of actions any one of which must be satisfied in order to satify the whole action. The system provided default method for OR predications returns this type of action description. The second element of the action description must be a list of action descriptions, i.e. lists returned by calling **joshua:expand-backward-rule-action**.

4. `(:procedure` *lisp-expression name*`)` This action description informs the rule compiler that the current trigger is not a pattern to be **joshua:ask**'ed but rather a Lisp expression that appears in the If-part of the backward rule. Such expressions are executed once all proceeding actions in the rule have been satisfied. The expression can act as a filter by returning either **joshua::t** or **joshua::nil**. **joshua::t** indicates success; in this case the bindings accumulated up to this point are considered acceptable and rule execution continues. **joshua::nil** indicates failure; in this case the bindings are considered unacceptable.

   The expression can also act as a generator in which it produces several new sets of bindings each of which is consistent with the bindings that were in effect just before the action began execution. To do this it should bind whatever logic-variables it wants to and then call **joshua:succeed**. **joshua:succeed** takes a rest-argument; the rule compiler will arrange for this value passed to **joshua:succeed** to be bound to the logic-variable which is the third element of the action description.

   See the function **joshua:succeed**, page 232.

5. `(:ignore)` This action description informs the rule compiler that it should ignore this action. The are two reasons for using this type of action description. The first is to allow a rule to have actions included in it simply for the sake of clarity. The second is to include actions only to specify context.

7.3.1. Customizing the Matchers Generated by the Rule Compiler

When Joshua compiles unifier functions for a pattern of a forward chaining rule, it actually compiles two procedures: a full unifier and a semi-unifier. (The latter assumes there are no variables on the data side of the match; when applicable, the second one is faster.)

In some cases, the **joshua:map-over-forward-rule-triggers** method will have already checked some slots against the data before handing it off to the unifier. In such a case, it would be silly for the unifier procedure to check the same features that have just been checked by **map-over-forward-rule-triggers**. The **joshua:positions-forward-rule-matcher-can-skip** protocol function is the hook that lets you advise the rule compiler about such situations. The rule compiler will then generate a semi-unification matcher that doesn't bother to check the slots identified by **joshua:positions-forward-rule-matcher-can-skip**. (The full unifier must still check all slots, because of the possibility of variables on the data side.)

In the default implementation, **joshua:positions-forward-rule-matcher-can-skip** instructs the match compiler for forward rules to generate a semi-unification matchers that ignores those parts of a predication that contain symbols. This is because the default trigger indexing scheme, as implemented by **(joshua:map-over-forward-rule-triggers joshua:default-protocol-implementation-model)** is a discrimination network that has already discarded candidates with different symbols in this position before we ever reached the unification question. (Trigger patterns

such as [good-to-eat honey], [good-to-eat bread], [good-to-eat goat-cheese] discriminate to different nodes.)

Suppose the trigger mapping method you use is the default one, which uses a discrimination net. When you give it a piece of data to find triggers for, it looks in the discrimination net and comes back with triggers that might unify with it. "Might unify", in this context, means that it has checked some things that are cheap to check, and used those to reject candidates that have no hope of unifying. In particular, the discrimination net checks all positions in a predication that have symbols, so there's no need to have the unifier check them again.

Here's an example. The default trigger indexing scheme uses a discrimination net, so the semi-matcher (for forward rules) can skip looking at any slots in the predication that contain symbols. Thus the default implementation of **joshua:positions-forward-rule-matcher-can-skip** could have been:

```
(define-predicate-method (positions-forward-matcher-can-skip predication)
                         ()
  (loop for token = (predication-statement self)
                then (cdr token)
        while (consp token)
        ;; needn't deal with tail variable, since variables
        ;; can't ever be skipped anyway
        when (symbolp (car token)) collect token))
```

and thus, if foo uses the default trigger indexing scheme:

```
(positions-forward-matcher-can-skip [foo a ?x b])
→ ((FOO A ?X B) (A ?X B) (B))
```

Note that the result returned is a list of tails of the predication.

If you create a customized trigger index, you have to be sure that you either inherit the right **joshua:positions-forward-rule-matcher-can-skip** method, or that you write a method of your own that is appropriate for your indexing scheme. In the above example, inheriting the symbol-skipping behavior from **joshua:default-rule-compilation-model** was correct. If you don't provide for the symbol-skipping behavior, you can run into situations where the unifier gives false matches. For instance, you could **joshua:tell** [foo a b] and have a forward rule whose trigger is [foo a bar] incorrectly fire. Needless to say, this can lead to very subtle and hard-to-analyze bugs. If you want to be very careful (and perhaps overly pessimistic) you can instruct the semi-matcher for forward chaining to check everything, by doing the following:

```
(define-predicate-method
 (positions-forward-matcher-can-skip <your-predicate-model-here>) ()
  nil)
```

This pessimistic method will ensure your semi-matchers are always correct; sometimes you can do better in terms of performance by considering what your trigger-indexing scheme has already looked at.

8. The Joshua Object Facility

8.1. Introduction to the Joshua Object Facility

A very large part of what one wants to express about the world is captured by object-attribute-value triples. For example:

- The color (attribute) of Fred's eyes (object) is blue (value);
- My checking account (object) has a balance (attribute) of $514.54 (value);
- The voltage (attribute) at Node-22 (object) is 27.2 (value).

It is often convenient to aggregate all the attributes of a particular object into a single data-structure. Object-Oriented programming systems such as Flavors (or the Common Lisp Object System) provide a natural mechanism for aggregating the various properties of an object into a single representation (e.g. a Flavor instance).

The Joshua Object Modelling facility unifies the object-oriented paradigm of the Flavors system with the Joshua rule-based paradigm. It does this by using the Joshua Protocol to map Object-Attribute-Value predications into the Object-oriented storage of the Flavor System. While the facilities of the Joshua Protocol have always made this possible, this release provides the capability as a built-in facility.

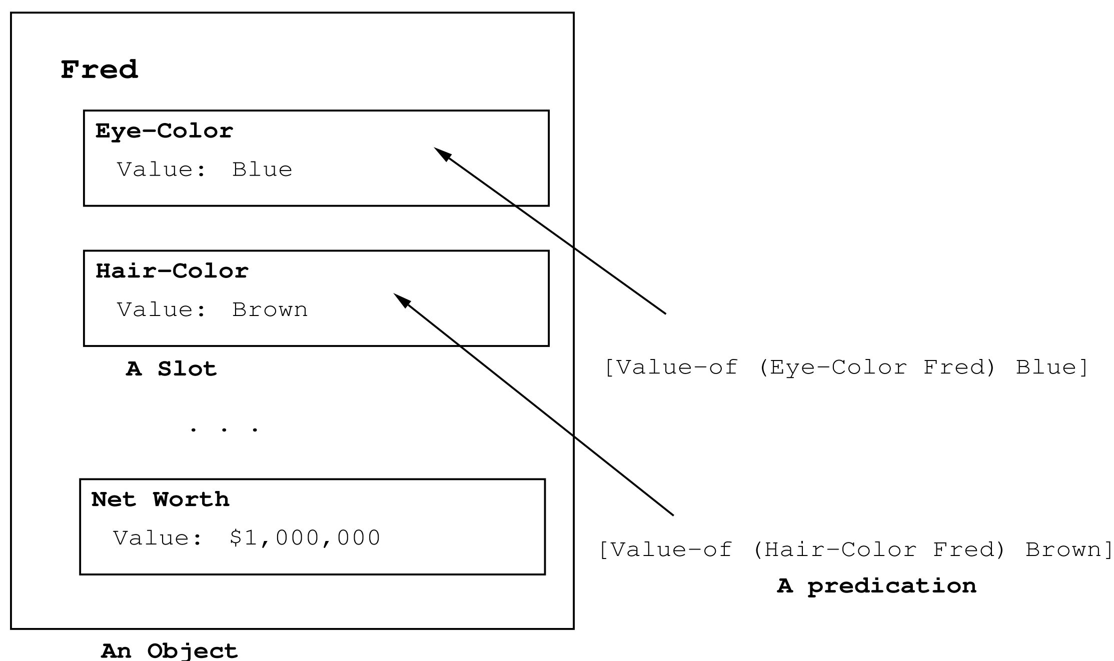


Figure 35. Predications Being Mapped into an Object Representation

The new facility also provides access to other common facilities of object-oriented systems. For example, when one TELL's

[value-of (Fred eye-color) Blue]

the new facility makes it possible to invoke an appropriate method associated with the class of the object named Fred or to invoke an expression association specifically with the Fred object.

Finally, the new facility makes it possible to create equality links between properties of different objects, making it easy to express an idea such as Fred's eye-color is the same as Sam's.

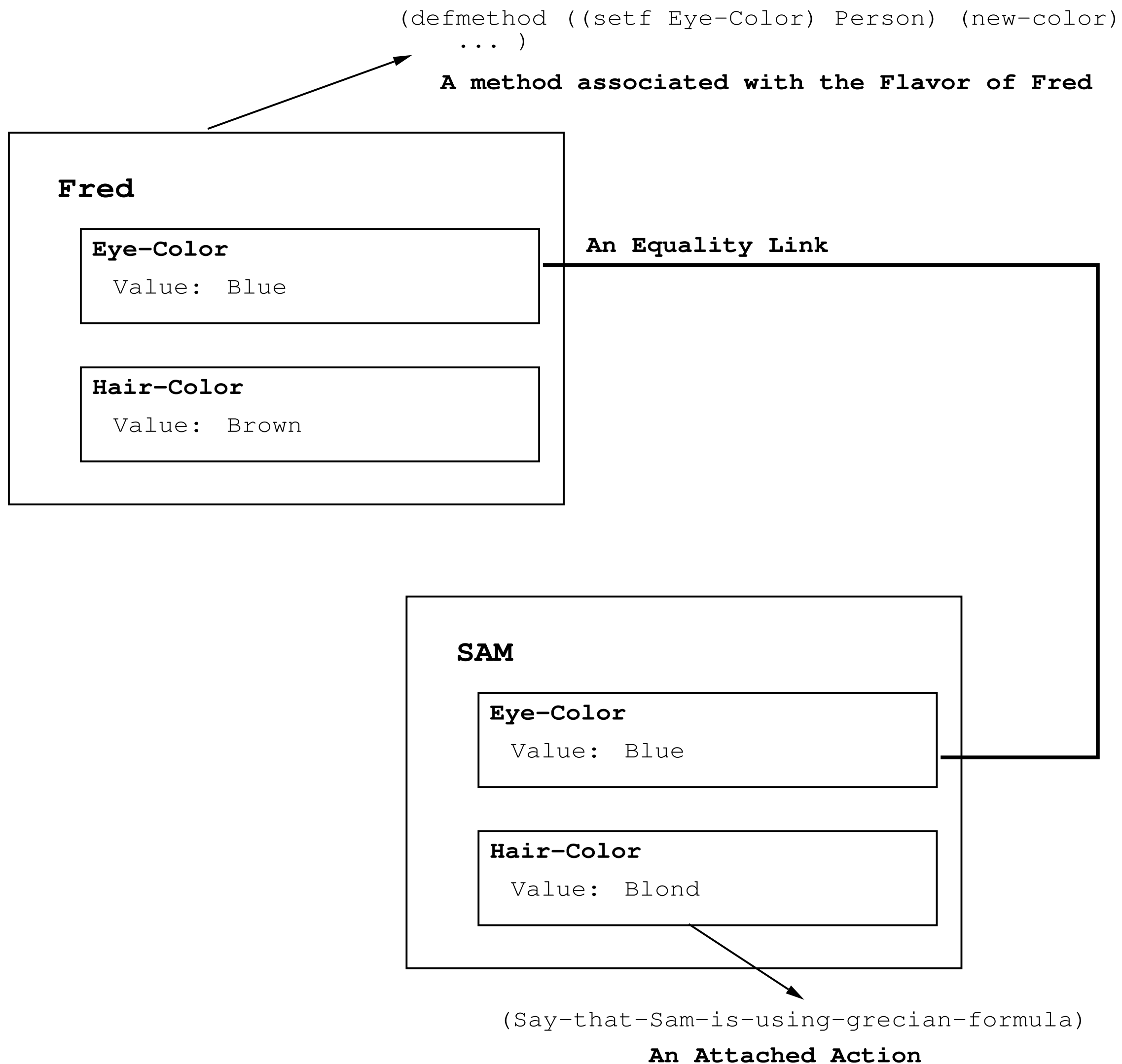


Figure 36. Other Capabilities of the Object Facility

8.2. Basic Capabilities of the Joshua Object Facility

Objects are represented in the Joshua Object facility as particular types of Flavors. (Every Joshua Object-Type includes a Flavor called Basic-Object as a component Flavor). Attributes of Joshua Objects are represented by Slots which are data structures attached to the Object. The type of a Joshua Object is called its Object-Type.

Objects and Slots provide a broad range of facilities; however, initially we will look only at the basic facilities.

Suppose we want to define an Object-Type to represent electrical resistors; this is done as follows:

```
(define-object-type resistor
  :slots (current voltage resistance))
```

This defines the Resistor Object-Type; any object of this type has three slots. Each of these has a field containing the actual value of the attribute (e.g. the voltage across the resistor) and other fields (such as a predication).

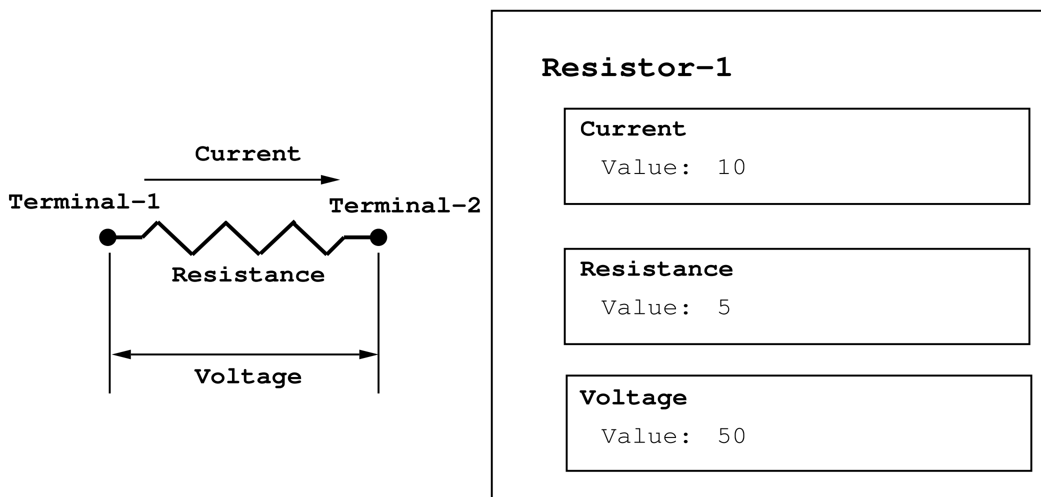


Figure 37. A Resistor and its Representation as an Object

An object is created using **joshua:make-object**:

```
(Setq Resistor-1 (make-object 'resistor :name 'resistor-1))
```

There is an accessor function corresponding to each slot of the object which returns the value of that attribute. Thus, the value of the current through the Resistor-1 can be retrieved as:

```
(Resistor-1 Current)
```

just as if we were retrieving the value of an instance variable. [However, the value of the Current instance-variable of Resistor-1 is actually the slot data structure;

the Current accessor function first fetches the slot and then extracts the value from that. Later on we will see how to fetch the slot as opposed to the value of the slot].

The predication **joshua:value-of** is Joshua's means for talking about slots and Objects. This predication takes two arguments, the first of which describes a slot and the second of which a value. For example,

```
[value-of (Resistor-1 voltage) 10]
```

says that Resistor-1's voltage is 10 (volts presumably).

Such predications can be used like any other; they can be used in **joshua:ask** and **joshua:tell**. To tell the system that Resistor-2 has a voltage of 5, we would say:

```
[value-of (Resistor-2 voltage) 5].
```

To determine the voltage across Resistor-2, we would might:

```
(ask [value-of (Resistor-2 voltage) ?voltage]
     #'(lambda (ignore)
        (print ?voltage)))
```

[Note: the first argument to a **joshua:value-of** predication can be either an expression describing a slot as shown in the examples above, or it can be an actual slot. Since we haven't yet shown how to fetch a slot, we'll postpone further mention of this for a while.]

A second predication used to talk about Joshua objects is **joshua:object-type-of**. This is an **ju::ask-only** predication, it can never be used as an argument to **joshua:tell**. **joshua:object-type-of** takes two arguments, the first of which is an Object and the second is the name of an Object-Type. Thus,

```
(ask `[Object-Type-Of ,Resistor-2 Resistor]
     #'(lambda (ignore)
        (print 'yes)))
```

queries whether Resistor-2 is, in fact, a resistor. While

```
(ask `[Object-Type-Of ,Resistor-2 ?His-type]
     #'(lambda (ignore)
        (print ?His-type)))
```

retrieves and prints the Object-Type of Resistor-2.

Finally, Joshua's rules can use these predications to express inferences that should be drawn. Continuing with our example of a resistor, we can express Ohm's law as follows:

```
(defrule Ohm (:forward)
  If [and [Object-type-of ?resistor Resistor]
          [value-of (?resistor current) ?I]
          [value-of (?resistor resistance) ?R]]
  Then `[value-of (?resistor voltage) ,(* ?I ?R)])
```

[Note: there is an important restriction on the use of these predications in rules. Any rule which includes a **joshua:value-of** predication must also contain an

joshua:object-type-of predication which describes the object-type of the object mentioned in the **joshua:value-of** predication. For questions about **joshua:value-of** predications, an **joshua:object-type-of** predication must be given as the **:context** option:

```
(defquestion Get-Resistor-Current (:backward)
             [value-of (?resistor current) ?value]
             :context [Object-type-of ?resistor Resistor])
```

Of course, we can also write backward-chaining rules which talk about objects. The rule above could have been written as:

```
(defrule Ohm (:backward)
  If [and [Object-type-of ?resistor Resistor]
          [value-of (?resistor current) ?I]
          [value-of (?resistor resistance) ?R]
          (unify ?voltage (* ?I ?R))]
  Then `[value-of (?resistor voltage) ?voltage])
```

8.3. Using Paths to Refer to the Structure of an Object

joshua:value-of predications normally refer to a slot (or an object) using Paths. A path is simply a list of names which describes how to find an object or a slot. We have already seen paths in the examples above. For example, in:

```
[value-of (Resistor-1 voltage) 10]
```

where the path (Resistor-1 voltage) describes the voltage slot of Resistor-1.

In general, objects contain other objects (for example, a computer-console has a screen, a keyboard, and a mouse; A mouse has a left, a middle and a right button). This leads to longer paths, such as:

```
(Howies-monitor mouse left-button up-down-state)
```

which describes the up-down-state attribute of the left-button of the mouse of Howie's monitor. We will see how to make such compound objects in a later section; until then, these longer paths will not be very important.

Each term in a path describes a subpart or slot of the object described by preceding terms in the path. One might, therefore, wonder what contains the first thing in a path. The answer is that there is a special, hidden object which is the root of the part-whole hierarchy; when we make an object it becomes a subpart of this root. Thus,

```
(make-object 'resistor :name 'r1)
```

makes an object whose object-type is Resistor and which is described by the path

```
(R1).
```

Its resistance is described by the path:

```
(R1 Resistance)
```

8.4. Type Hierarchy in the Joshua Object Facility

Object-types can include other Object-Types (just as a Flavor can mix in other Flavors). For example, all resistors, capacitors and inductors are Two Terminal Devices; all such devices share certain properties (for example, they all have two terminals, a voltage across them and a current through them). It is a useful (and modular) to capture the common features in a single type definition which is then shared by the subordinate types. We can do this as follows:

```
(define-object-type 2-terminal-device
  :slots (current terminal-1-current terminal-2-current
          voltage terminal-1-voltage terminal-2-voltage))

(define-object-type resistor
  :slots (resistance)
  :included-object-types (2-terminal-device))

(define-object-type capacitor
  :slots (capacitance)
  :included-object-types (2-terminal-device))

(define-object-type capacitor
  :slots (inductance)
  :included-object-types (2-terminal-device))
```

Which says that all 2-terminal devices have 2 terminals, of which has a voltage and a current. There is a voltage across any 2-terminal device and a total current through any 2-terminal device. Resistors, have all these properties; in addition a resistor has a resistance; similarly for capacitors and capacitance and inductors and inductance.

This modularity is particularly useful, because we can take advantage of it in our rules. We can express the fact that the voltage across any 2-terminal device is the difference between its two terminal voltages with a simple rule:

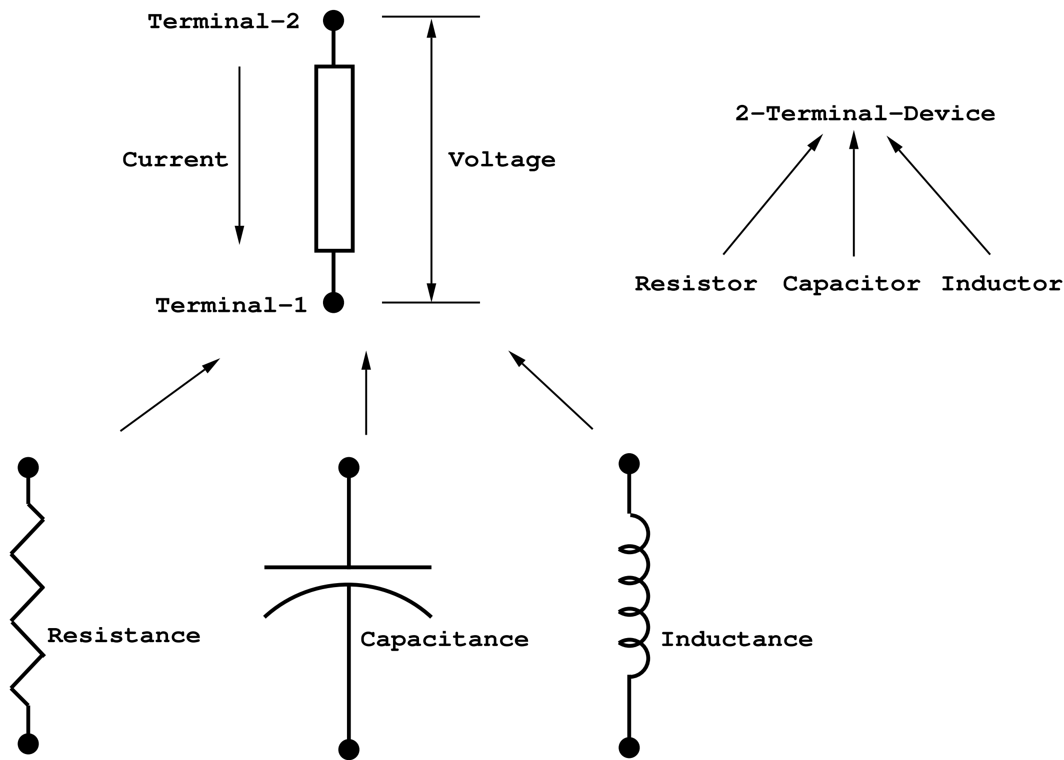


Figure 38. The Object-Type Hierarchy of Two-Terminal Devices

```
(defrule 2-terminal-voltage (:forward)
  If [and [Object-type-of ?device 2-terminal-device]
          [value-of (?device terminal-1-voltage) ?t1-voltage]
          [value-of (?device terminal-2-voltage) ?t2-voltage]]
  Then `[value-of (?device voltage) ,(- ?t2-voltage ?t1-voltage)])
```

This rule, however, will apply to any 2-terminal device (whether it is a resistor, capacitor or inductor). So if we were to create a resistor and state its two terminal voltages, this rule would fire:

```
(make-object 'resistor :name r1)

(tell [value-of (r1 terminal-1-voltage) 0])
(tell [value-of (r1 terminal-2-voltage) 5])
```

and deduce the new predication:

```
[Value-Of (r1 voltage) 5]
```

This same rule would also apply to a capacitor, for example:

```
(make-object 'capacitor :name C1)
(tell [value-of (C1 terminal-1-voltage) 0])
(tell [value-of (C1 terminal-2-voltage) 5])
```

which would lead to the deduction of:

```
[Value-Of (C1 voltage) 5]
```

Of course, the Ohm's law rule would apply only to resistors.

Notice that since R1 is both a resistor and a 2-terminal device, there are 2 valid answers to a query about its Object-Type. Thus, the query:

```
(ask [Object-Type-Of R1 ?Type]
     #'(lambda (ignore)
         (print ?Type)))
```

will print both Resistor and 2-terminal-device. Similary, if we query for all objects of the 2-terminal-device object-type, as follows:

```
(ask [Object-Type-Of ?thing 2-terminal-device]
     #'(lambda (ignore)
         (print ?Thing)))
```

we will get a listing of all resistors, capacitors and inductors.

8.5. Part-Whole Hierarchy in the Joshua Object Facility

Objects often fall into a second natural hierarchy, that of part inclusion. (See the section "Using Paths to Refer to the Structure of an Object", page 109.)

For example, a "Widget" factory might have three parts: a warehouse for receiving incoming material, a widget milling station and a warehouse for storing completed widgets waiting shipment. Each of these is an object which must, therefore, have an object-type of its own. The two warehouses are objects of the Warehouse object-type and the milling station is an object of the Milling-Machine object-type. In addition, each of these objects plays a particular role within the widget company. Notice that, although there are two warehouses, they play different different roles and, therefore, have different names within the context of a Widget factory. **joshua:define-object-type** takes a **:parts** keyword argument whose value is a list of pairs of roles and object-types. Thus, we would describe a widget factory, as follows:

```
(define-object-type widget-factory
  :parts ((receiving warehouse)
          (production milling-machine)
          (finished-goods warehouse)))
```

This says that every widget-factory has one subpart named *Receiving* which is a warehouse and another subpart named *Finished-Goods* which is also a warehouse. Finally, any Widget-Factory also has a part named *Production*, which is a Milling-Machine.

Notice that we can describe any of these using paths. We can create a widget-factory, named WF-1 as follows:

```
(make-object 'widget-factory :name 'WF-1)
```

Its three subparts can be refered to with the following paths:

```
(WF-1 Milling-Machine)
(WF-1 Receiving)
(WF-1 Finished-Goods)
```

Assume that an object of type Milling-Machine has a slot named Thing-Being-Milled. Then the path

```
(WF-1 Milling-Machine Thing-Being-Milled)
```

names the thing currently being milled in the Production part of WF-1.

8.6. Other Capabilities of Slots

Slots support a variety of behaviors other than those explained already. This behavior is controlled by keyword arguments attached to the name of the slot in its Define-Object-Type form. If no keywords are specified (as has been the case so far) then the default behaviors are assumed.

8.6.1. Initial Values of Slots

The initial value of a slot may be specified by including the :initform keyword argument in the slot description. The value of this argument is a form; the slot is initialized to have the value of this form as its contents.

For example,

```
(define-object-type resistor
 :slots ((resistance :initform 10)))
```

specifies that whenever a resistor is created, its resistance should be initialized to 10. This is done by causing the appropriate **joshua:tell** to happen.

8.6.2. Set Valued and Single Valued Slots

Slots may be either single-valued or set-valued. If a slot is single-valued, then at any one time it there can only be a single value of the attribute represented by the slot. For example, the voltage at node-22 can only be a specific voltage at any time. It would be contradictory to believe both:

```
[value-of (voltage node-22) 10]
[value-of (voltage node-22) 20].
```

However, other attributes can be set valued; for example, there might be many siblings of John. Here it is perfectly possible to believe both of:

```
[value-of (sibling john) mary]
[value-of (sibling john) mark]
```

joshua:define-object-type specifies for each slot whether it is single-valued or set-valued. Single-valued is the default. The fact that sibling is a set-valued attribute would be indicated by:

```
(define-object-type person
   :slots ((sibling :set-valued t)
           ...))
```

8.6.3. Slots and Truth Maintenance

Another option in the description of a slot, is whether the information in the slot should be subject to truth-maintainence or not. (The default is no). If the slot is subject to truth-maintenance then any predication mentioning it should include the **ltms::ltms-mixin.** The provided predicate **ltms:value-of** includes both slot modelling and LTMS mixins. Suppose that we wanted to reason about Adders as part of trouble-shooting program which uses Truth-Maintenance techniques (such a program is included in the Jericho demo suite). We would define Adder as follows:

```
(define-object-type adder
   :slot ((status :truth-maintenance t)
          (addend :truth-maintenance t)
          (augend :truth-maintenance t)
          (sum :truth-maintenance t)))
```

And we would **joshua:tell** that the value of the addend of adder-22 is 10 as follows:

```
(tell [ltms:value-of (adder-22 addend) 10]).
```

[Note: It would be an error to use the predicate Value-of].

8.6.4. Slots and Attached Actions

Another feature supported by slots is the ability to attach a Lisp expression to a particular slot which is triggered whenever the value is changed. This is specified by the **:attached-actions** keyword (the default is no attached actions). If we wanted to enable the ability to attach such Lisp expressions to the SUM of adders we would specify:

```
(define-object-type adder
   :slot ((status :truth-maintenance t)
          (addend :truth-maintenance t)
          (augend :truth-maintenance t)
          (sum :truth-maintenance t :attached-actions t)))
```

This allocates extra space in the SUM slot of every adder to hold the attached lisp expression. The attached action is simply a Lisp function which is run every time the slot changes value. To actually attach an action, the function Add-Action is used:

```
(add-action '(adder-22 sum) #'print-pathname-and-value)

(defun print-pathname-and-value (cell current-value predication ignore)
  (if (eql (predication-truth-value predication) *true*)
      (format t "~&The value of ~s is ~s" cell current-value)
      (format t "~&The value of ~s isn't ~s any more"
              cell current-value)))
```

joshua:add-action takes two required arguments and one optional argument. The first argument is either a path to a slot or an actual slot and the second argument is the function to attach to that slot. The function is called for side effect; when invoked, it is passed the following arguments: the slot, the current-value of the slot, the predication associated with the value of the slot, and the previous truth-value of the predication. The function is called whenever the slot assumes a new value or whenever a current value is removed; the truth-value of the predication argument and the previous truth-value can be used to distinguish the two cases (as shown above).

The optional argument to **joshua:add-action** is a "name" for the action. It defaults to **:action**. Naming an action allows the user to attach more than one action to a slot. Individual actions may be removed from a slot by using **joshua:remove-action**.

Attached actions can be used for a variety of purposes: For example, they can be used to implement validity checks on the values inserted in a slot, or they can be used in set-valued slots to check that the cardinality of the set is within some bounds.

8.6.5. Invoking Methods Associated with the Object Associated with a Slot

A final capability of slots is the ability to invoke a method associated with the object with with the slot is associated whenever the value of the slot changes. [Note: This is different than an attached-action (see the section "Slots and Attached Actions", page 114) in that an attached-action is associated with a particular object while a method is associated with every object of the type.] For example, if we wanted every Adder to notice the changing of its status attribute, we would indicate this as follows:

```
(define-object-type adder
   :slot ((status :truth-maintenance t :object-notifying t)
          (addend :truth-maintenance t)
          (augend :truth-maintenance t)
          (sum :truth-maintenance t :attached-actions t)))
```

When the status attribute of an adder changes, Joshua will call the (SETF STATUS) method of the ADDER flavor. The arguments passed are: the value in the slot and the predication associated with the slot. Thus, if the **:object-notifying** option were specified as above, one would be expected to define a method such as the following:

```
(defmethod ((setf status) adder) (current-value current-predication)
  (when (and (eql current-value :broken)
             (eql (predication-truth-value predication) *true*))
    (Sound-the-alarm self)))
```

which checks to see that the current state of the adder is :BROKEN and if so, sounds its alarm.

8.6.6. Equalities Between Slot Values

When an object is decomposed into a sub-part hierarchy, it often happens that there are connections between certain attributes of the sub-parts. For example, a "Voltage Divider" is a simple electrical circuit consisting of two connected resistors. The voltage at terminal-1 of one of the two resistors will necessarily be equal to the voltage at terminal-2 of the other resistor.

There will be similar connections in a factory, where the output of one machine is the input of another, or in a accounting system where the result of one calculation is an input to another. **joshua:define-object-type** provides a means of expressing such equalities as follows:

```
(define-object-type voltage-divider
   :slots (voltage current output-voltage terminal-1-voltage terminal-2-voltage)
   :parts ((resistor-1 resistor)
           (resistor-2 resistor))
   :equalities (((resistor-1 current) (resistor-2 current))
                ((resistor-1 current) (current))
                ((resistor-1 terminal-2-voltage) (resistor-2 terminal-1-voltage))
                ((resistor-1 terminal-1-voltage) (terminal-1-voltage))
                ((resistor-2 terminal-2-voltage) (terminal-2-voltage))))
```

which, in effect, specifies the wiring diagram shown in Figure 39.

If two slots are specified to be equal, then any time the value of one of the slots is determined, the value of the other will automatically be deduced.

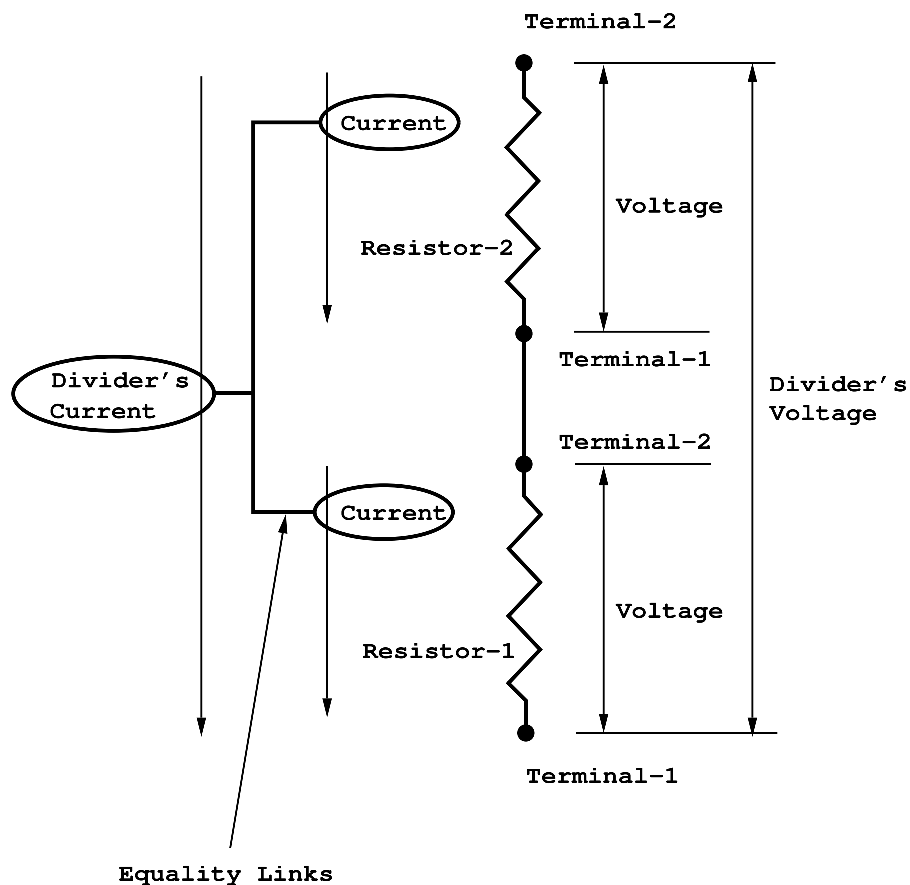


Figure 39. Equality Links in a Two Resistor Voltage Divider

8.7. Other Options in Define-Object-Type

A Joshua Object is, in fact, a Flavor Instance. It is often useful to be able to include in Joshua Objects instance variables which do not hold slots as their values. Similarly, it is useful to be able to mix a normal flavor into an Object-Type definition. The syntax of Define-Object-Type supports this, using the **:other-instance-variables** and **:other-flavors** keyword arguments. For example,

```
(Define-Object-Type voltage-divider
   :slots (voltage current output-voltage terminal-1-voltage terminal-2-voltage)
   :parts ((resistor-1 resistor)
           (resistor-2 resistor))
   :other-instance-variables (documentation)
   :other-flavors (electronic-component-with-documentation-mixin)
   )
```

This has the effect of mixing the electronic-component-with-documentation-mixin flavor into the voltage-dividers flavor. In addition, every instance of the voltage-divider object-type will have a "documentation" instance variable.

Finally, it is sometimes useful to be able to include some initialization code as part of the Define-Object-Type form. This can be done using the **:initializations** keyword argument. The value of this argument is a list of Lisp forms which are run after an instance of the type is created. In effect, this code is appended to the the **make-instance** method of the Object-Type; thus, self is bound to the created object while this code executes. For example, the following definition

```
(define-object-type manufacturing-site
   :slots (production-capacity
           clock
           input-request
           input-request-acknowledge
           (output :attached-actions t))
   :other-instance-variables ((things-being-produced (make-heap)))
   :initializations ((tell `[value-of (,self output) NIL]))
   )
```

initializes the output slot of manufacturing sites to be **nil**.

8.8. The Predicates Used in the Joshua Object Facility

There are four Predicates used in the Joshua Object facility. Besides Define-Object-Type, these are the main interfaces to the facility. Each of these predicates is supplied in two forms: The first mixes in the standard Joshua TMS capabilities (**ltms::ltms-predicate-mixin**) while the second omits TMS capabilities (and is based on **joshua:default-predicate-model**). The basic capabilities of each of the four predicates is supplied as a Predicate-Model which can be combined with user supplied Predicate-Models to provide whatever capabilities you need.

The four predicates are:

Purpose	*Without TMS*	*With TMS*	*Model mixin*
Slot Accessing	**value-of**	**ltms:value-of**	**slot-value-mixin**
Part-Whole	**part-of**	**ltms:part-of**	**part-of-mixin**
Object-Type	**object-type-of**	**ltms:object-type-of**	**type-of-mixin**
Equality	**equated**	**ltms:equated**	**equated-mixin**

joshua:part-of and **joshua:object-type-of** are ask-only predicates; they cannot be used in **joshua:tell**. This is because the information they refer to is the unchangeable information of the part-whole and object-type hierarchies created by **joshua:define-object-type**. **joshua:value-of** is the normal way of accessing the values of slots; we have seen examples of its use above. **joshua:equated** can be used to create equalities not specified by **joshua:define-object-type**. For example:

```
(let ((warehouse1 (make-object 'warehouse :name 'warehouse1))
      (factory1 (make-object 'factory-1 :name 'factory1)))
  (tell `[equated (warehouse1 output) (factory1 input)]))
```

connects the output slot of Warehouse1 to the input slot of Factory1.

9. Joshua Language Dictionary

9.1. Dictionary Entries

joshua:act-on-truth-value-change *database-predication old-truth-value* *Generic Function*

database-predication A predication

old-truth-value The truth value that just changed

Called whenever the truth-value of *predication* changes from *old-truth-value* to some new truth-value. The new truth-value is available in the predication by the time act-on-truth-value-change is called. It can be examined using **joshua:predication-truth-value**.

This protocol function allows you to take actions that depend on the truth value of a predication as the truth values change. (You might want to do that, for example, in advanced uses of modeling.)

When a predication changes truth-value, the TMS may make several other predications to change their truth-values as well. The TMS is responsible for first calling **joshua:notice-truth-value-change** on every changed predication before this protocol function is invoked. Thus whenever an **joshua:act-on-truth-value-change** method is called, it may safely assume that the world has been updated into a consistent state.

See the sections on "Signalling Truth Value Changes" and **joshua:notice-truth-value-change**

joshua:add-action *slot-or-path function* &optional *(name* **:action***)* *Generic Function*

This function is part of the Joshua object facility. It allows actions, which are arbitrary functions, to be associated with a slot of a Joshua object. The function will be called whenever the value of a slot changes.

The function is called for side effect; when invoked, it is passed the following arguments: the slot, the current-value of the slot, the predication associated with the value of the slot, and the previous truth-value of the predication. The function is called whenever the slot assumes a new value or whenever a current value is removed; the truth-value of the predication argument and the previous truth-value can be used to distinguish the different possibilities.

Actions may be removed from slots by using **joshua:remove-action**.

The optional argument to **joshua:add-action** is a "name" for the action. It defaults to **:action**. Naming an action allows the user to attach more than one action to a slot.

joshua:add-backward-question-trigger *predication truth-value trigger-object context question-name* — *Generic Function*

predication	The pattern under which the backward question is to be indexed.
truth-value	The truth value under which the pattern is to be indexed.
trigger-object	The backward question trigger data-structure to be indexed.
context	The context of the backward question. Useful in advanced modeling applications.
question-name	The name of the backward question being indexed.

Tailoring of backward-question indexing is usually accomplished by providing methods for the **joshua:locate-backward-question-trigger** and **joshua:map-over-backward-question-triggers** protocol functions. The **joshua:add-backward-question-trigger** and **joshua:delete-backward-question-trigger** methods provided as Joshua's defaults call **joshua:locate-backward-question-trigger** as a subroutine. All of the interesting tailoring of their behavior can be obtained by providing a **joshua:locate-backward-question-trigger** method.

However, it might be useful in some applications to provide **:before** or **:after** methods for the add and delete methods, for example to keep track of what questions are currently indexed where. Even in advanced modeling applications it is unlikely that you will need to define a method for the **joshua:add-backward-question-trigger** protocol function.

See the section "The Joshua Question Indexing Protocol", page 48.

joshua:add-backward-rule-trigger *predication truth-value trigger-object context rule-name* — *Generic Function*

predication	The pattern under which the backward rule is to be indexed.
truth-value	The truth value under which the backward rule is to be indexed.
trigger-object	The backward rule trigger data-structure to be indexed.
context	The entire *if*-part of the rule. Useful in advanced modeling applications.
rule-name	The name of the rule being indexed.

Tailoring of backward rule indexing is usually accomplished by providing methods for the **joshua:locate-backward-rule-trigger** and **joshua:map-over-backward-rule-triggers** protocol functions. The **joshua:add-backward-rule-trigger** and **joshua:delete-backward-rule-trigger** methods provided as

Joshua's defaults call **joshua:locate-backward-rule-trigger** as a subroutine. All of the interesting tailoring of their behavior can be obtained by providing a **joshua:locate-backward-rule-trigger** method.

However, it might be useful in some applications to provide **:before** or **:after** methods for the add and delete methods, for example to keep track of what rules are currently indexed where. Even in advanced modeling applications it is unlikely that you will need to define a method for the **joshua:add-backward-rule-trigger** protocol function.

See the section "The Contract of the Trigger Adding Functions", page 38.

joshua:add-forward-rule-trigger *predication truth-value trigger-object context rule-name* — *Generic Function*

predication	The pattern under which the forward rule is to be indexed.
truth-value	The truth value under which the pattern is to be indexed.
trigger-object	The forward rule trigger data-structure to be indexed.
context	The entire *if*-part of the rule. Useful in advanced modeling applications.
rule-name	The name of the rule being indexed.

Tailoring of forward rule indexing is usually accomplished by providing methods for the **joshua:locate-forward-rule-trigger** and **joshua:map-over-forward-rule-triggers** protocol functions. The **joshua:add-forward-rule-trigger** and **joshua:delete-forward-rule-trigger** methods provided as Joshua's defaults call **joshua:locate-forward-rule-trigger** as a subroutine. All of the interesting tailoring of their behavior can be obtained by providing a **joshua:locate-forward-rule-trigger** method.

However, it might be useful in some applications to provide **:before** or **:after** methods for the add and delete methods, for example to keep track of what rules are currently indexed where. Even in advanced modeling applications it is unlikely that you will need to define a method for the **joshua:add-forward-rule-trigger** protocol function.

See the section "The Contract of the Trigger Adding Functions", page 38.

joshua:ask *query continuation* &key *(:do-backward-rules* **t***) :do-questions* — *Function*

Queries the virtual database and backward rules and questions.

Note: joshua:ask is a macro, and as such it cannot be used as an argument to the function **funcall**.

query	Should be a predication.

continuation Should be a function of one argument, describing what you want done with the answers to the query.

Note that the argument given to *continuation* might be ephemeral in one of two ways: it could be stack-consed, and it could contain logic variables whose bindings will be undone when you exit this frame. Instantiated queries almost always need to be copied with **joshua:copy-object-if-necessary**, because the variable bindings are ephemeral. See example 6 below.

If, on the other hand, you are collecting database predications, they are not ephemeral, and you don't want to copy them. (Copying a database predication causes loss of the database information associated with the predication.)

Keywords:

:*do-backward-rules* If this keyword has a non-**nil** value, backward chaining rules are checked for solutions. The default is **t**. Use :*do-backward-rules* **nil** to check out just the database solution.

:*do-questions* If this keyword has a non-**nil** value, any questions that claim to answer *query* are run to solicit more solutions from the user. The default is **nil**.

joshua:ask uses the database, backward rules, and questions to satisfy the query predication. Each time **joshua:ask** finds a solution to *query* it calls the continuation, passing it a list that contains the answer and information about how the answer was derived.

joshua:ask doesn't return an interesting value. Normally the continuation performs some action with each solution. You can collect values in the continuation, or return a value to some caller of **joshua:ask** using **throw**, **return-from**, or some similar Lisp form. Such uses of **throw** and **return-from** are like the Prolog **cut** feature. See examples 6 through 9.

Any logic variables used in *query* can be referred to as though they were lexical Lisp variables within *continuation*; **joshua:ask** establishes a binding contour for the logic variables. (See example 1 below.) In this sense, **joshua:ask** is like **let** combined with **mapc**. Like **let**, **joshua:ask** establishes lexical binding contours for the logic variables in the query. Like **mapc**, it iteratively calls the continuation on the answer. For a discussion of scoping rules: See the section "Variables and Scoping in Joshua" in *User's Guide to Basic Joshua*.

joshua:ask calls the continuation function with a single argument, *backward-support*, a list containing information about the solution process. The list contains the instantiated query, its truth value, and the support for the query; the form of the support varies, depending on how the query was satisfied.

Typically you'll want to deal only with part of the information provided in *backward-support* rather than with the entire list. For instance, you might want to see only the answer, or only the database predication that matched the answer, or only the support for the answer.

Joshua supplies *accessor functions* to extract various elements of the list in *backward-support*, making it available to you for interpretation.

In addition, Joshua provides *convenience functions* that extract some element of the list in *backward-support* and interpret it for you. These functions let you postpone dealing with the details of *backward-support* and accessor functions until you need them for more advanced work. So before reading on you might want to skip ahead to the section "Streamlining Typical Continuation Requests with Convenience Functions" and see if these functions meet your current needs.

Continuation Argument

backward-support A list of the following form:

- The first element is always the unified query, that is, the query that was passed to **joshua:ask**, with appropriate variables instantiated as a side-effect of unification.

- The second element is the truth value of the query. This corresponds to the truth value of the matching predication in the database at the time **joshua:ask** looked at it.

- The rest of the elements are the support for the instantiated query. The support can take several forms, depending on how the query was satisfied.

  - When the query is satisfied by matching a predication in the database, the support is that database object.

  - When the query answer comes from a conjunction (**and**), the support is the symbol **and**, followed by the backward support for each of the compound predications.

  - When the query answer comes from a disjunction (**or**), the support is the symbol **or**, followed by the support for the single predication from the **or** that succeeded.

  - When the query answer is derived from a backward rule, the support has the format

    ```
    ((rule rule-name) . rule-support)
    ```

    where

- *rule* is the symbol rule
- *rule-name* is the name of the rule used to satisfy the query
- *rule-support* is a list containing (recursively) the backward support used to satisfy parts of the rule body.

° When the query answer comes from a question, the support is like that for rules, except that it uses the question name instead of the rule name.

° When the query answer comes from the predicates **joshua:known** or **joshua:provable**, the support is the respective symbol name (**joshua:known** or **joshua:provable**), followed by the support for the predication that served as the symbol's argument.

° When the query originates from an **joshua:ask** or an **joshua:ask-data** method, the support is whatever the writer of that method provided. See the section "Customizing the Data Index", page 81.

In schematic form, the *backward-support* list looks as follows:

```
The backward-support list:
(<unified query> <truth-value> . <derivation>)

(<(unified) query>)

(<t/f>)

(<derivation>)  Possibilities for these elements are:
  (<database predication>)
  (AND <conjunct1 derivation> <conjunct2 derivation> ...)
  (OR <successful disjunct derivation>)
  ((RULE <rule name>) <conjunct1 t/f derivation> <conjunct2 t/f derivation> ...)
  ((QUESTION <question name>) <succeed argument>)
  (KNOWN <derivation>)
  (PROVABLE <derivation>)
```

Extracting Parts of the Continuation with Accessor Functions

Joshua provides four accessor functions to extract specific portions of *backward-support*. Use these functions if you want to interpret the answer yourself. Use the convenience functions described below if you want the system to interpret the information for you.

joshua:ask-query Extracts the instantiated query (the first element) from *backward-support*. For example:

```
(ask [...] #'(lambda (backward-suppport)
               (print (ask-query backward-support))))
```

joshua:ask-query-truth-value

Extracts the truth value of the instantiated query (the second element) from *backward-support*. For example:

```
(ask [...] #'(lambda (backward-support)
               (print
                 (ask-query-truth-value backward-support))))
```

joshua:ask-database-predication

Extracts the database object that matched *query*. If the backward support is a rule, displays the rule name (see example 4). Use this function only when you know the support is a database object (that is, with **:do-backward-rules nil**. For example:

```
(ask [...]
  #'(lambda (backward-support)
      (print (ask-database-predication backward-support)))
  :do-backward-rules nil)
```

joshua:ask-derivation

Extracts the support information in *backward-support*. Makes fewer assumptions than **joshua:ask-database-predication** about where the support came from. For example:

```
(ask [...] #'(lambda (backward-support)
               (print (ask-derivation backward-support))))
```

Streamlining Typical Continuation Requests with Convenience Functions

When an **joshua:ask** query succeeds, there are some standard things you might want to do with the answer, such as: printing or formatting the unified query, operating on the database predication supporting the query, or interpreting all of the backward support.

Joshua provides five convenience functions that extract an appropriate part of the answer and interpret it in some specific way. The first four are **joshua:ask** continuation functions. The fifth is a special-purpose function that lets you do database lookup only, and interpret the answer in some way. **joshua:map-over-database-predications** uses **joshua:ask** to search the database and extract the predication(s) matching its argument pattern.

These functions are:

joshua:print-query Extracts and displays the unified query. For example:

```
(ask [...] #'print-query)
```

joshua:say-query Extracts the unified query and displays it in formatted form.

joshua:print-query-results
Takes the information in *backward-support* and displays it with annotations.

joshua:graph-query-results
The above in graph form.

joshua:map-over-database-predications
For special cases of the solution process, where you look only in the database for an answer, extracts all database predications that unify with a predication pattern and applies some function to each. For example:

```
(map-over-database-predications [foo ?x] #'untell)
```

joshua:map-over-database-predications is equivalent to:

```
(ask query #'(lambda (x) (funcall continuation
                                  (ask-database-predication x)))
     :do-backward-rules nil)
```

We use some of the convenience functions in the examples to **joshua:ask**. For more on each function, please consult its dictionary entry.

Examples of Using joshua:ask

Let's define some predicates, enter them into the database, then add a backward rule and a backward question. The rule determines what is an eater's favorite food. The question elicits information to satisfy the rule's subgoal.

```
(define-predicate favorite-meal (eater food))
(define-predicate guzzles (eater food))

(defun eat-it ()
  (clear)
  (tell [and [favorite-meal bears honey]
             [favorite-meal mosquitoes people]
             [favorite-meal spiders flies]
             [favorite-meal monkeys bananas]
             [guzzles ted ice-cream]])
  (cp:execute-command "Show Joshua Database"))
```

```
Show Joshua Database
 True things
  [FAVORITE-MEAL BEARS HONEY]
  [FAVORITE-MEAL MOSQUITOES PEOPLE]
  [FAVORITE-MEAL SPIDERS FLIES]
  [FAVORITE-MEAL MONKEYS BANANAS]
  [GUZZLES TED ICE-CREAM]
 False things
  None

(defrule not-finicky (:backward)
  if [guzzles ?eater ?food]
  then [favorite-meal ?eater ?food])

(defquestion guzzler? (:backward)
  [guzzles ?eater ?food])
```

Next we **joshua:ask** what Joshua knows about everybody's favorite meals. Example 1 uses the variables in the unified query to print an English-like sentence (not fussy about number agreement between subject and verb) about everybody's meals. It ignores the *backward-support* argument and uses a **format** directive. It looks in the database and rules, but not in questions.

```
Example 1.
(ask [favorite-meal ?eater ?food]
     #'(lambda (ignore)
         (format t "~%~S is the preferred food of ~S." ?food ?eater)))
BANANAS is the preferred food of MONKEYS.
FLIES is the preferred food of SPIDERS.
PEOPLE is the preferred food of MOSQUITOES.
HONEY is the preferred food of BEARS.
ICE-CREAM is the preferred food of TED.
```

Example 2 prints the instantiated query for everybody's meals, using the convenience function, **joshua:print-query**. It uses the database only, ignoring both rules and questions.

```
Example 2.
(ask [favorite-meal ?eater ?food] #'print-query :do-backward-rules nil)
  ;print just those in the database
[FAVORITE-MEAL MONKEYS BANANAS]
[FAVORITE-MEAL SPIDERS FLIES]
[FAVORITE-MEAL MOSQUITOES PEOPLE]
[FAVORITE-MEAL BEARS HONEY]
```

Example 3 prints the instantiated query for the meals of bears, using the convenience function, **joshua:print-query**. It looks in the database and backward rules, but not in questions.

```
Example 3.
(ask [favorite-meal bears ?food] #'print-query)
  ;print out bears' favorite-meal foods
[FAVORITE-MEAL BEARS HONEY]
```

Example 4 prints the predication object that satisfied the query for everybody's meals using the accessor function **joshua:ask-database-predication**. It looks in the database and backward rules, but not in questions. Notice that when the query is satisfied from a rule, the rule name is printed, not a predication object. It is best to use **joshua:ask-database-predication** with **:do-backward-rules nil**, that is, when you know the support is only in the database.

```
Example 4.
(ask [favorite-meal ?eater ?food]
     #'(lambda (backward-support)
          (print (ask-database-predication backward-support))))
[FAVORITE-MEAL MONKEYS BANANAS]
[FAVORITE-MEAL SPIDERS FLIES]
[FAVORITE-MEAL MOSQUITOES PEOPLE]
[FAVORITE-MEAL BEARS HONEY]
(RULE NOT-FINICKY)
```

Example 5 prints the instantiated query for everybody's meals. It uses the database, backward rules, *and* questions. Note that we supplied just one answer interactively to the question, although we could have supplied more.

```
Example 5.
(ask [favorite-meal ?eater ?food] #'print-query :do-questions t)
  ;look for backward questions as well


For what values of ≡EATER and ≡FOOD is it true that "[GUZZLES ≡EATER ≡FOOD]"?
Some solution exists: Yes  No
Value for ≡EATER: CHRISTOPHER
Value for ≡FOOD: BANANA-PIE
<ABORT> aborts, <END> uses these values

[FAVORITE-MEAL CHRISTOPHER BANANA-PIE]
What are some more values of ≡EATER and ≡FOOD such that "[GUZZLES ≡EATER ≡FOOD]"?
Some solution exists: Yes  No
<ABORT> aborts, <END> uses these values
NIL
➪


[FAVORITE-MEAL MONKEYS BANANAS]
[FAVORITE-MEAL SPIDERS FLIES]
[FAVORITE-MEAL MOSQUITOES PEOPLE]
[FAVORITE-MEAL BEARS HONEY]
[FAVORITE-MEAL TED ICE-CREAM]
[FAVORITE-MEAL CHRISTOPHER BANANA-PIE]
```

Example 6 collects a list of patterns that describe everybody's meals. It uses the database and rules, but not questions. Note the use of **joshua:copy-object-if-necessary**. This is because the bindings in the query are undone

upon exit from the continuation, so we must make a copy in which to preserve them.

Note that the resulting list is *not* a list of things that are in the database, but rather a list of free-floating predications that are copies of the query. If you want the latter, use **joshua:ask-database-predication** with **:do-backward-rules nil** and don't copy it. See example 7.

```
Example 6.
(defun collect-answers ()
  (let ((answers nil))
    (ask [favorite-meal ?eater ?food]
         #'(lambda (backward-support)
             (push (copy-object-if-necessary
                     (ask-query backward-support)) answers)))
    answers))
COLLECT-ANSWERS

  (collect-answers)
([FAVORITE-MEAL TED ICE-CREAM] [FAVORITE-MEAL BEARS HONEY]
 [FAVORITE-MEAL MOSQUITOES PEOPLE]
 [FAVORITE-MEAL SPIDERS FLIES] [FAVORITE-MEAL MONKEYS BANANAS])
```

Example 7 is identical to example 6, except that here we collect database predications instead of instantiated queries, and the former don't need to be copied. Since we are only looking in the database we specify **:do-backward-rules nil**.

```
(defun collect-answers-database-predications ()
  (let ((answers nil))
    (ask [favorite-meal ?eater ?food]
         #'(lambda (backward-support)
             (push (ask-database-predication backward-support)
                   answers)
         :do-backward-rules nil))
    answers))
COLLECT-ANSWERS-DATABASE-PREDICATIONS

(collect-answers-database-predications)
([FAVORITE-MEAL BEARS HONEY]
 [FAVORITE-MEAL MOSQUITOES PEOPLE]
 [FAVORITE-MEAL SPIDERS FLIES] [FAVORITE-MEAL MONKEYS BANANAS])
```

Better style for the above example would be:

```
(collect-answers-database-predications2 ()
  (let ((answers nil))
    (map-over-database-predications [favorite-meal ?eater ?food]
                                    #'(lambda (db-predication)
                                        (push db-predication answers)))
    answers))
```

Often you're interested in whether there *is* a solution, but not any *particular* solution. Example 8 illustrates the use of **return-from** in a continuation to return when the first solution is found.

```
Example 8.
(defun solution-exists-p ()
  (ask [favorite-meal ?eater ?food]
       #'(lambda (ignore)
           (return-from solution-exists-p t)))
    ;; return nil if nothing succeeded
    nil))

   (solution-exists-p)
T
```

Example 9 is like the example above, but it returns a copy of the query, instead of a boolean. This is useful if you want to know something about the solution, in addition to its existence. (However, if you want to use database-related properties, such as TMS-relation, use **joshua:ask-database-predication** and don't copy it).

```
Example 9.
(defun first-solution ()
  (block find-a-solution
    (ask [favorite-meal ?eater ?food]
         #'(lambda (backward-support)
             (return-from find-a-solution
               (copy-object-if-necessary (ask-query backward-support)))))
    ;; return nil if nothing succeeded
    nil))

  (first-solution)
[FAVORITE-MEAL MONKEYS BANANAS]
```

Modeling Note:

Chances are that you seldom want to define a method that takes over the entire functionality of **joshua:ask**. It's more likely you want to define a method for one of the generic functions it calls, such as **joshua:fetch**, **joshua:ask-data**, **joshua:ask-rules**, **joshua:ask-questions**, or **joshua:map-over-forward-rule-triggers**.

Also, there is a **sys:downward-funarg** declaration on *continuation*, so your implementations of **joshua:ask** should not use *continuation* in other than stack-like ways.

Related Functions:

joshua:tell
joshua:clear
joshua:copy-object-if-necessary
joshua:map-over-database-predications

See the section "Querying the Database" in *User's Guide to Basic Joshua*. See the section "The Joshua Database Protocol", page 8. See the section "Customizing the Data Index", page 81.

joshua:ask-data *predication truth-value continuation* *Generic Function*

predication	A predication to search for.
truth-value	The truth value being asked about. Must be either **joshua:*true*** or **joshua:*false***.
continuation	A function to be called when the data is found.

joshua:ask-data is the database part of the **joshua:ask** protocol. It is an intermediate level of the protocol, between **joshua:ask** and **joshua:fetch**. It is called by the default **joshua:ask** method, and the default method for **joshua:ask-data** calls **joshua:fetch**. You will probably not call this function directly, except when writing **joshua:ask** methods. More commonly, you might write your own **joshua:ask-data** method as a kind of data modeling.

The complete contract of **joshua:ask-data** is:

- Look in the virtual database for *predication* or anything which unifies with it.
- Make sure that the current truth value of the entry in the database matches *truth-value*.
- Unify *self* with a copy of the database predication.
- (Assuming all has gone well so far) build the appropriate backward support and call *continuation* with the backward support.

The backward support for an **joshua:ask-data** method should be a list of three elements:

- *self*, the (now unified) query predication.
- *truth-value*, the truth value being **joshua:ask**ed.
- The derivation. This will usually be the database predication. If there is no database predication, this should be some other indication of the derivation of this query succcess. Typically, this would be a symbol indicating the reason for success.

Actually, if your model is storing the database predications as predication objects, you probably don't need to write an **joshua:ask-data** method. Writing your own **joshua:fetch** method and using the default **joshua:ask-data** method is more convenient. Defining an **joshua:ask-data** method is usually done when you don't want to actually store the predication objects. See the

first version of the `good-to-eat` model in the section"Customizing the Data Index" for an example of this.

See the generic function **joshua:ask-rules**, page 142. See the generic function **joshua:ask-questions**, page 140.

joshua:ask-data-and-questions-only-mixin *Flavor*

This flavor defines an **joshua:ask** method that only looks in the database and asks questions (if **:do-questions** is non-**nil**), but never tries backward rules.

The default **joshua:ask** method looks first in the database, then tries backward rules (if **:do-backward-rules** is non-**nil**), then asks questions (if **:do-questions** is non-**nil**).

This flavor can be used as a component of a predicate or of a predicate model to change how **joshua:ask** is implemented for that predicate or for predicates of that model. To use it, specify it as a component in **joshua:define-predicate** or **joshua:define-predicate-model**. The flavor is provided primarily as a means of increasing the performance of **joshua:ask** by skipping protocol steps which are not needed for some particular predicates.

Related Flavors:

joshua:default-ask-model
joshua:ask-data-only-mixin
joshua:ask-rules-only-mixin
joshua:ask-questions-only-mixin
joshua:ask-data-and-rules-only-mixin
joshua:ask-rules-and-questions-only-mixin

joshua:ask-data-and-rules-only-mixin *Flavor*

This flavor defines an **joshua:ask** method that only looks in the database and tries backward rules, but never asks questions.

The default **joshua:ask** method looks first in the database, then tries backward rules (if **:do-backward-rules** is non-**nil**), then asks questions (if **:do-questions** is non-**nil**).

This flavor can be used as a component of a predicate or of a predicate model to change how **joshua:ask** is implemented for that predicate or for predicates of that model. To use it, specify it as a component in **joshua:define-predicate** or **joshua:define-predicate-model**. The flavor is provided primarily as a means of increasing the performance of **joshua:ask** by skipping protocol steps which are not needed for some particular predicates.

Related Flavors:

joshua:default-ask-model
joshua:ask-data-only-mixin
joshua:ask-rules-only-mixin
joshua:ask-questions-only-mixin
joshua:ask-data-and-questions-only-mixin
joshua:ask-rules-and-questions-only-mixin

joshua:ask-database-predication *backward-support* *Function*

An accessor function for use in an **joshua:ask** continuation. It extracts the database predication that matched the query from the continuation argument, *backward-support*, that contains information about the satisfied query. We describe this continuation argument fully in the dictionary entry for **joshua:ask**.

Note that if the backward support did not come from the database, **joshua:ask-database-predication** gives a bogus answer; in some cases, such as user-written models, it may even cause a trip to the debugger. Thus, you should use **joshua:ask-database-predication** only with **:do-backward-rules nil**.

Examples:

We build a library database using **joshua:tell** statements as well as a forward rule that says the library owns any work authored by Shakespeare. We also include an LTMS in our predicate definitions so that we can later apply **joshua:explain** to the database predications we find.

```
(define-predicate author-of (work author) (ltms:ltms-predicate-model))
(define-predicate owns-library (work) (ltms:ltms-predicate-model))

(defrule Shakespeare-holdings (:forward)
  if [author-of ?work Shakespeare]
  then [owns-library ?work])

(defun build-author-title-index2 ()
  (clear)
  (tell [and [author-of "King Lear" Shakespeare]
             [author-of "Hedda Gabler" Ibsen]
             [owns-library "Trumpeting Joshua"]
             [author-of "A Doll's House" Ibsen]])
  (cp:execute-command "Show Joshua Database"))
BUILD-AUTHOR-TITLE-INDEX2

(build-author-title-index2)
True things
  [OWNS-LIBRARY "Trumpeting Joshua"]  [AUTHOR-OF "Hedda Gabler" IBSEN]
  [OWNS-LIBRARY "King Lear"]          [AUTHOR-OF "King Lear" SHAKESPEARE]
  [AUTHOR-OF "A Doll's House" IBSEN]
False things
  None
```

Now we ask Joshua to find and **joshua:explain** the database predications that tell what the library owns.

```
(ask [owns-library ?work]
     #'(lambda (backward-support)
         (explain (ask-database-predication backward-support))))
[OWNS-LIBRARY "Trumpeting Joshua"] is *True*.
  It's a :Premise.
[OWNS-LIBRARY "King Lear"] is *True*.
  It's derived from the rule Shakespear-Holdings, using:
  [AUTHOR-OF "King Lear" SHAKESPEARE]
```

Usually you can use the convenience function **joshua:map-over-database-predications** instead of **joshua:ask-database-predication**.

For comparison we use the same library example for both functions.

For more on these and related functions: See the function **joshua:ask**, page 123.

joshua:ask-data *truth-value continuation* of **joshua:default-ask-model** *Method*

This is the default **joshua:ask-data** method. It does something like the following (somewhat sanitized) code:

```
(define-predicate-method (ask-data default-ask-model)
                         (truth-value continuation)
  (fetch self
         #'(lambda (database-predication)
             (when (= truth-value
                      (predication-truth-value database-predication))
               ;; the truth value we're looking for matches the
               ;; database predication
               (with-unification
                 ;; if the database predication has variables, copy it
                 ;; so the database isn't side-effected
                 (unify self
                        (copy-object-if-necessary database-predication))
                 ;; the unification succeeded, so call the continuation
                 (stack-let ((backward-support `(,self
                                                 ,truth-value
                                                 ,database-predication)))
                   (funcall continuation backward-support)))))))
```

joshua:ask-data-only-mixin *Flavor*

This flavor defines an **joshua:ask** method thatwhich only looks in the database, and never tries rules or questions.

The default **joshua:ask** method looks first in the database, then tries backward rules (if **:do-backward-rules** is non-**nil**), then asks questions (if **:do-questions** is non-**nil**).

This flavor can be used as a component of a predicate or of a predicate model to change how **joshua:ask** is implemented for that predicate or for predicates of that model. To use it, specify it as a component in **joshua:define-predicate** or **joshua:define-predicate-model**. The flavor is provided primarily as a means of increasing the performance of **joshua:ask** by skipping protocol steps which are not needed for some particular predicates.

Related Flavors:

joshua:default-ask-model
joshua:ask-rules-only-mixin
joshua:ask-questions-only-mixin
joshua:ask-data-and-rules-only-mixin
joshua:ask-data-and-questions-only-mixin
joshua:ask-rules-and-questions-only-mixin

joshua:ask-derivation *backward-support* *Function*

An accessor function for use in an **joshua:ask** continuation. It extracts the support information about the satisfied query from the continuation argument *backward-support*.

Note that the accessor function **joshua:ask-database-predication** makes more assumptions about the support than **joshua:ask-derivation** does.

Here is a schematic representation of the contents of *backward-support*. **joshua:ask-derivation** extracts only the derivation portion. For more detail please consult the dictionary entry for **joshua:ask**.

```
The backward-support list:
(<unified query> <truth-value> . <derivation>)

(<(unified) query>)

(<t/f>)

(<derivation>)  Possibilities for these elements are:
  (<database predication>)
  (AND <conjunct1 derivation> <conjunct2 derivation> ...)
  (OR <successful disjunct derivation>)
  ((RULE <rule name>) <conjunct1 t/f derivation> <conjunct2 t/f derivation> ...)
  ((QUESTION <question name>) <succeed argument>)
  (KNOWN <derivation>)
  (PROVABLE <derivation>)
```

Like the other accessor functions, **joshua:ask-derivation** does not interpret the information it extracts. Generally you won't need to use it very often.

Note that the convenience functions **joshua:print-query-results** and **joshua:graph-query-results**, respectively, display and graph an annotated version of the support information, so that you don't have to interpret it yourself.

For comparison we'll use the same examples to illustrate all three of these functions.

Examples:

The first example shows the support for a query satisfied by database lookup — the database predication that satisfied the query is printed.

```
(define-predicate type-of (object type))

(tell [type-of Iliad epic])

Example 1.
(ask [type-of ?book epic]
  #'(lambda (backward-support)
       (print (ask-derivation backward-support))))
([TYPE-OF ILIAD EPIC])
```

The next example shows the support for a query that is satisfied from rules. We have a rule, dessert?, that determines if a given food is a dessert. Each of this rule's subgoals is derived from other rules. Here are the definitions.

```
; Example 2. Query is derived from backward rules
; Define the predicates
(define-predicate edible (object))
(define-predicate is-food (object))
(define-predicate contains (object substance))
(define-predicate sweet (object))

; Define the rules
(defrule food? (:backward)
  if [edible  ?object]
  then [is-food  ?object])

(defrule sweet? (:backward)
  if [or [contains ?object chocolate]
         [contains ?object sugar]
         [contains ?object honey]]
  then [sweet ?object])

(defrule dessert? (:backward)
  if [and [is-food ?object]
          [sweet ?object]]
  then [type-of ?object dessert])

; tell some sticky facts
(tell [edible chocolate-coated-ants])
(tell [contains chocolate-coated-ants honey])
```

Now we **joshua:ask** what foods qualify as desserts and why. The display is

a list starting with rule `dessert?` that satisfied the query; next is the first subgoal that was satisfied, together with its truth value, and the name of the rule which satisfied it (rule `food?`). That rule's first subgoal is then listed with its truth value and the database predication that satisfied it, and so on, through all the backward support.

```
(ask [type-of ?object dessert]
  #'(lambda (backward-support)
       (print (ask-derivation backward-support))))
((RULE DESSERT?)
 ([IS-FOOD CHOCOLATE-COATED-ANTS] 1 (RULE FOOD?)
  ([EDIBLE CHOCOLATE-COATED-ANTS] 1 [EDIBLE CHOCOLATE-COATED-ANTS]))
 ([SWEET CHOCOLATE-COATED-ANTS] 1 (RULE SWEET?)
  ([CONTAINS CHOCOLATE-COATED-ANTS HONEY] 1
                                   [CONTAINS CHOCOLATE-COATED-ANTS HONEY])))
```

For more on these and related functions: See the function **joshua:ask**, page 123.

joshua:ask-query *backward-support* *Function*

An accessor function for use inside an **joshua:ask** continuation. It extracts the instantiated query from the continuation argument *backward-support*.

backward-support is fully described in the dictionary entry for **joshua:ask**.

Example:

Here we collect and save all the answers from a query. (See example 6 in the dictionary entry for **joshua:ask**.)

```
(defun collect-answers ()
  (let ((answers nil))
    (ask [favorite-meal ?eater ?food]
         #'(lambda (backward-support)
              (push (copy-object-if-necessary
                        (ask-query backward-support))
                    answers)))
    answers))
```

To extract and print out the instantiated query, use the convenience function **joshua:print-query**.

For more on these and related functions: See the function **joshua:ask**, page 123.

joshua:ask-query-truth-value *backward-support* *Function*

An accessor function for use inside an **joshua:ask** continuation. It extracts the truth value of the instantiated query from the continuation argument *backward-support*.

backward-support is fully described in the dictionary entry for **joshua:ask**.

The truth value is a number, as follows:

0	Truth value of **joshua:*unknown***
1	Truth value of **joshua:*true***
2	Truth value of **joshua:*false***
3	Truth value of **joshua:*contradictory***

The **joshua:truth-value** presentation type translates these numbers into symbols naming a truth value.

Most of the time you know the query's truth value from the query pattern itself, so that you have little need of this function. The truth value information is mostly there for system use, to let the system interpret the query.

Examples:

```
(define-predicate status-of (object status))
(tell [status-of smoke-alarm off])

; Example 1.
(ask [status-of ?indicator off]
  #'(lambda (backward-support)
      (print (ask-query-truth-value backward-support))))
1

; Example 2. Use truth-value-name to translate the number
(ask [status-of ?indicator off]
  #'(lambda (backward-support)
      (print (truth-value-name (ask-query-truth-value backward-support)))))
*TRUE*
```

For more on this and related functions: See the function **joshua:ask**, page 123.

joshua:ask-questions *predication truth-value continuation* *Generic Function*

predication	A predication to be the goal for rules.
truth-value	The truth value being asked about. Must be either **joshua:*true*** or **joshua:*false***.
continuation	A function to be called when the rule is satisfied.

joshua:ask-questions is the question-asking part of the **joshua:ask** protocol. It is an intermediate level of the protocol, between **joshua:ask** and **joshua:map-over-backward-question-triggers**. It is called by the default **joshua:ask** method, and the default method for **joshua:ask-questions** calls **joshua:map-over-backward-question-triggers**. You will probably not call this function directly, except when writing **joshua:ask** methods. Since question compilation is not yet completely generic, you probably don't want to define your own **joshua:ask-questions** methods.

The complete contract of **joshua:ask-questions** is:

- Look in the question database for questions with a pattern predication matching *predication* or anything which unifies with it.
- Make sure that *truth-value* is appropriate. (If the question can be asked negatively, then either truth value is appropriate. If the question can only be asked positively, the truth value must be **joshua:*true***.)
- Unify *predication* (**self** in the method) with a copy of the pattern predication.
- Get the answer to the query in some appropriate way. If the query should succeed, call the *continuation* function.

The backward support for an **joshua:ask-questions** method should be a list containing:

- *predication* (**self** in the method), the (now unified) query predication.
- *truth-value*, the truth value being **joshua:ask**ed.
- (question *name*), where *name* is the name of the question.
- The derivation. This can be any extra information about how the question was answered or why it succeeded.

joshua:ask-questions-only-mixin *Flavor*

This flavor defines an **joshua:ask** method that only asks questions (if **:do-questions** is non-**nil**).

The default **joshua:ask** method looks first in the database, then tries backward rules (if **:do-backward-rules** is non-**nil**), then asks questions (if **:do-questions** is non-**nil**).

This flavor can be used as a component of a predicate or of a predicate model to change how **joshua:ask** is implemented for that predicate or for predicates of that model. To use it, specify it as a component in **joshua:define-predicate** or **joshua:define-predicate-model**. The flavor is provided primarily as a means of increasing the performance of **joshua:ask** by skipping protocol steps which are not needed for some particular predicates.

Related Flavors:

joshua:default-ask-model
joshua:ask-data-only-mixin
joshua:ask-rules-only-mixin
joshua:ask-data-and-rules-only-mixin
joshua:ask-data-and-questions-only-mixin
joshua:ask-rules-and-questions-only-mixin

joshua:ask-rules *predication truth-value continuation do-questions* *Generic Function*

predication A predication to be the goal for rules.

truth-value The truth value being asked about. Must be either **joshua:*true*** or **joshua:*false***.

continuation A function to be called when the rule is satisfied.

do-questions If non-**nil**, allow asking questions in subgoals of the rule.

joshua:ask-rules is the backward-chaining part of the **joshua:ask** protocol. It is an intermediate level of the protocol, between **joshua:ask** and **joshua:map-over-backward-rule-triggers**. It is called by the default **joshua:ask** method, and the default method for **joshua:ask-rules** calls **joshua:map-over-backward-rule-triggers**. You will probably not call this function directly, except when writing **joshua:ask** methods. Since rule compilation is not yet completely generic, you probably don't want to define your own **joshua:ask-rules** methods.

The complete contract of **joshua:ask-rules** is:

- Look in the rule database for rules with a *then*-part matching *predication* or anything which unifies with it.
- Make sure that the truth value of the *then*-part matches *truth-value*.
- Unify *predication* (**self** in the method) with a copy of the *then*-part.
- **joshua:ask** the *if*-part. In the continuation of the **joshua:ask** build the appropriate backward support and call the *continuation* of **joshua:ask-rules** with the backward support.

The backward support for an **joshua:ask-rules** method should be a list containing:

- *predication* (**self** in the method), the (now unified) query predication.
- *truth-value*, the truth value being **joshua:ask**ed.
- (rule *name*), where *name* is the name of the rule.

- The derivation. This will be a list of the support for the *if*-part.

See the section "Finding Backward Rule Triggers", page 43.

joshua:ask-rules-and-questions-only-mixin *Flavor*

This flavor defines an **joshua:ask** method that only tries backward rules (if **:do-backward-rules** is non-**nil**) and asks questions (if **:do-questions** is non-**nil**), but never looks in the database.

The default **joshua:ask** method looks first in the database, then tries backward rules (if **:do-backward-rules** is non-**nil**), then asks questions (if **:do-questions** is non-**nil**).

This flavor can be used as a component of a predicate or of a predicate model to change how **joshua:ask** is implemented for that predicate or for predicates of that model. To use it, specify it as a component in **joshua:define-predicate** or **joshua:define-predicate-model**. The flavor is provided primarily as a means of increasing the performance of **joshua:ask** by skipping protocol steps which are not needed for some particular predicates.

Related Flavors:

joshua:default-ask-model
joshua:ask-data-only-mixin
joshua:ask-rules-only-mixin
joshua:ask-questions-only-mixin
joshua:ask-data-and-rules-only-mixin
joshua:ask-data-and-questions-only-mixin

joshua:ask-rules-only-mixin *Flavor*

This flavor defines an **joshua:ask** method that only tries backward rules (if **:do-backward-rules** is non-**nil**).

The default **joshua:ask** method looks first in the database, then tries backward rules (if **:do-backward-rules** is non-**nil**), then asks questions (if **:do-questions** is non-**nil**).

This flavor can be used as a component of a predicate or of a predicate model to change how **joshua:ask** is implemented for that predicate or for predicates of that model. To use it, specify it as a component in **joshua:define-predicate** or **joshua:define-predicate-model**. The flavor is provided primarily as a means of increasing the performance of **joshua:ask** by skipping protocol steps which are not needed for some particular predicates.

Related Flavors:

joshua:default-ask-model
joshua:ask-data-only-mixin
joshua:ask-questions-only-mixin
joshua:ask-data-and-rules-only-mixin
joshua:ask-data-and-questions-only-mixin
joshua:ask-rules-and-questions-only-mixin

joshua:basic-tms-mixin *Flavor*

This flavor must be included in any TMS model predicate. It does not define any of the TMS protocol methods itself, but it ensures that TMS predicates support the correct protocol methods.

joshua:clear &optional (*clear-database* **t**) (*undefrule-rules* **nil**) *Function*

With arguments **t t**, empties the database and "undoes" all rule definitions.

clear-database	Specifies whether or not to clear the database. Default is **t**.
undefrule-rules	Specifies whether or not to delete all rule definitions. Default is **nil**.

Clearing the database is equivalent to **joshua:untell**ing each fact in the database.

Note that undefining all rule definitions is a drastic thing to do, as it clears out *all* rules in your world. Any application depending on these rules will no longer work. Clear out all rules only if you want a "clean" environment, for example, if you need to get rid of a runaway rule that you cannot stop by other means.

Examples:

```
Show Joshua Database
 True things
  [FAVORITE-MEAL BEARS HONEY]
  [FAVORITE-MEAL MOSQUITOES PEOPLE]
  [FAVORITE-MEAL SPIDERS FLIES]
  [FAVORITE-MEAL MONKEYS BANANAS]
 False things
  None

(clear)

Show Joshua Database
 True things
  None
 False things
  None
```

Related Command:

"Clear Joshua Database Command"

See the section "Removing Predications From the Database" in *User's Guide to Basic Joshua*.

See the section "The Joshua Database Protocol", page 8.

See the section "Customizing the Data Index", page 81.

Clear Joshua Database Command

Clears predications from the Joshua Database.

Predications — Which predications to remove from the database. Clear Joshua Database asks the database for all predications matching those specified in the *Predications* argument and **joshua:untell**s them from the database. The value of *Predications* can also be All or None.

:*Other Truth Values Too* — Whether or not to clear the predications in the database which match those specified by the *Predications* argument, but have the opposite truth value. This argument defaults to Yes.

:*Query* — Whether to ask you before making changes to the database. By default, the command stops and asks before removing any predications or rules.

:*Undefine Rules* — If *Undefine Rules* is Yes, the command will undefine all of the Joshua Rules. This argument defaults to No.

:*Verbose* — Whether to print information about what the command is doing.

Clear Joshua Database provides a convenient interface to the **joshua:untell** function. It asks the database for all predications matching those specified by the arguments, prompts you for confirmation, and **joshua:untell**s each predicate. It also allows you to undefine all the Joshua rules, resulting in a fresh Joshua environment.

Note that undefining all rule definitions is a drastic thing to do, as it clears out *all* rules in your world. Any application depending on these rules will no longer work. Clear out all rules only if you want a "clean" environment, for example, if you need to get rid of a runaway rule that you cannot stop by other means.

Related Functions:

joshua:clear
joshua:untell

joshua:*contradictory* *Variable*

A named constant used by Joshua to denote an interim state of computation wherein a predication is believed to be both **joshua:*true*** and **joshua:*false***. When this occurs, Joshua invokes the appropriate Truth Maintenance System to resolve the contradictory state.

joshua:*contradictory* is not meaningful unless a TMS is present. However, not all Truth Maintenance Systems are required to use this value.

Related Topics:

joshua:*true*
joshua:*false*
joshua:*unknown*
joshua:truth-value
joshua:predication-truth-value

See the section "Truth Values" in *User's Guide to Basic Joshua*. See the section "Justification and Truth Maintenance" in *User's Guide to Basic Joshua*.

joshua:copy-object-if-necessary *object* *Function*

Copies the *object* handed to it if it contains variables, or is otherwise ephemeral.

object	Any object, for example, a list, or a predication

Variables in *object* are renamed during copying, so that variables in the copy differ from variables in the original.

joshua:copy-object-if-necessary is useful for making copies of predications that may be stack-consed, or whose variables may be temporarily unified. The latter, for example, is true of variables in the query to **joshua:ask**.

joshua:copy-object-if-necessary creates a separate copy of its argument in the heap.

Examples: Here we reuse some of the examples introduced with **joshua:ask**. We define some predicates and a rule, then enter some facts into the database.

```
(define-predicate favorite-meal (eater food))
(define-predicate guzzles (eater food))

  (clear)
  (tell [and [favorite-meal bears honey]
             [favorite-meal mosquitoes people]
             [favorite-meal spiders flies]
             [favorite-meal monkeys bananas]
             [guzzles ted ice-cream]])
```

```
Show Joshua Database
 True things
  [FAVORITE-MEAL BEARS HONEY]
  [FAVORITE-MEAL MOSQUITOES PEOPLE]
  [FAVORITE-MEAL SPIDERS FLIES]
  [FAVORITE-MEAL MONKEYS BANANAS]
  [GUZZLES TED ICE-CREAM]
 False things
  None

(defrule not-finicky (:backward)
  if [guzzles ?eater ?food]
  then [favorite-meal ?eater ?food])

Example 1.
;;;If you don't copy the query, you lose the information!
(defun collect-answers-wrong ()
  (let ((answers nil))
    (ask [favorite-meal ?eater ?food]
         #'(lambda (backward-support)
             (push (ask-query backward-support) answers)))
    answers))
COLLECT-ANSWERS-WRONG

  (collect-answers-wrong)
#<Error printing object CONS 42353464>

Example 2.
;;;Using copy-object-if-necessary saves the information
(defun collect-answers ()
  (let ((answers nil))
    (ask [favorite-meal ?eater ?food]
         #'(lambda (backward-support)
             (push (copy-object-if-necessary
                     (ask-query backward-support)) answers)))
    answers))
COLLECT-ANSWERS

  (collect-answers)
([FAVORITE-MEAL TED ICE-CREAM] [FAVORITE-MEAL BEARS HONEY]
 [FAVORITE-MEAL MOSQUITOES PEOPLE]
 [FAVORITE-MEAL SPIDERS FLIES] [FAVORITE-MEAL MONKEYS BANANAS])
```

```
(defun first-solution ()
  (block find-a-solution
    (ask [favorite-meal ?eater ?food]
         #'(lambda (backward-support)
             (return-from find-a-solution
               (copy-object-if-necessary (ask-query backward-support)))))
    ;; return nil if nothing succeeded
    nil))
FIRST-SOLUTION

  (first-solution)
[FAVORITE-MEAL MONKEYS BANANAS]
```

Related Functions:

joshua:ask

joshua:database-predication *&key (print-truth-value t)* *Presentation Type*

Database-predication is a subtype of predication that includes only predications that have been in the database.

print-truth-value When *print-truth-value* is not nil, the predication is presented with a truth-value indicator (¬, ?, or ↔, that is, **not**, ***unknown***, ***contradictory***, respectively). By default the system prints truth-value indicators for all database predications.

This type is useful when defining commands or handlers that can apply only to predications that have been in the database. Joshua programs sometimes store extra information on predications as they are put in the database; TMS justification is one example. When a command depends on this information being present it should accept a database predication as its argument rather than just a predication.

database-predication can only accept input from the mouse, as that is the only way to uniquely identify a database predication.

Related Topic:

joshua:predication

joshua:default-ask-model *Flavor*

This flavor provides the Joshua default behavior for the **joshua:ask** protocol. It has methods for **joshua:ask**, **joshua:ask-data**, **joshua:ask-rules**, and **joshua:ask-questions**.

Related Flavor:

joshua:default-predicate-model

joshua:default-predicate-model *Flavor*

This is the default flavor for Joshua predications — the one you get when you don't specify any model to **joshua:define-predicate**. It defines all the default behavior for the Joshua protocol. It does not provide any TMS behavior.

This flavor is made up of **joshua:discrimination-net-data-mixin** and **joshua:default-protocol-implementation-model**.

Related topics:

ltms:ltms-predicate-model
joshua:define-predicate

joshua:default-protocol-implementation-model *Flavor*

This flavor provides the Joshua default behavior for the high-level protocol. It is composed of **joshua:default-rule-compilation-model**, **joshua:default-ask-model** and **joshua:default-tell-model**.

Related Function:

joshua:define-predicate-model

joshua:default-rule-compilation-model *Flavor*

This flavor defines all the default Joshua rule compilation behavior.

Related topics:

joshua:default-protocol-implementation-model
joshua:define-predicate-model

joshua:default-tell-model *Flavor*

This flavor provides the Joshua default behavior for the **joshua:tell** protocol. It has methods for **joshua:tell**, **joshua:insert**, **joshua:justify**, **joshua:notice-truth-value-change**, **joshua:unjustify**, and **joshua:untell**.

Related Flavor:

joshua:default-predicate-model

joshua:define-object-type *name* &key *:slots :parts :equalities :initializations :included-object-types :other-instance-variables :other-flavors* *Macro*

This macro is part of the Joshua object facility. It is used to define classes of objects.

name A symbol which will become the type-name of these objects.

:slots A list of slot descriptions, where each description is either a symbol which will become the slot-name, or a list consisting of a symbol followed by keyword-value pairs. Possible keywords are:

:initform Specifies an initial value for the slot. See the section "Initial Values of Slots", page 113.

:set-valued Specifies whether the slot is set-valued. See the section "Set Valued and Single Valued Slots", page 113. The value of this argument should be **t** to create a set-valued slot; it defaults to **nil** if not mentioned.

:truth-maintenance
Specifies whether the slot's values are maintained by the truth-maintenance system. The value of this argument should be **t** to create a truth-maintained slot; it defaults to **nil** if not mentioned. See the section "Slots and Truth Maintenance", page 114.

:attached-actions Specifies that the user may wish to add actions to individual instances of objects containing this slot. See the section "Slots and Attached Actions", page 114.

:object-notifying Specifies that the user intends to define a **setf** method on the slot value of the object type. See the section "Invoking Methods Associated with the Object Associated with a Slot", page 115.

```
(define-object-type random-machine
     :slots (oil-viscosity
             (gears :set-valued t)
             (fuel-volume :attached-actions t
                          :truth-maintenance t)
             ...)
     ...)
```

:parts A list of part descriptions, where each description is a list containing a part-name and the type of the part.

:equalities A list of equality descriptions, where each description is a list of pathnames relative to the object being created. See the section "Equalities Between Slot Values", page 116.

:initializations A list of forms to evaluate at make-instance time. See the section "Other Options in Define-Object-Type", page 117.

:included-object-types
Specifies a list of other Joshua-object-types from which to inherit. See the section "Type Hierarchy in the Joshua Object Facility", page 110.

:other-instance-variables
A list of regular flavor instance variables to be included in the object definition. See the section "Other Options in Define-Object-Type", page 117.

:other-flavors A list of other flavors to mix in to the object definition. See the section "Other Options in Define-Object-Type", page 117.

joshua:define-predicate *name args* &optional *(model-and-other-components* **'(default-predicate-model)***)* &body *options* *Macro*

Defines a predicate for use in building predications.

name Any symbol that does not conflict with the name of an existing flavor or presentation type. So, for example, **integer**, **cons**, and **array** are not good predicate names. In fact, they can be disastrous. Doing **joshua:define-predicate** on these will likely cause problems in both the CL type system and the presentation system.

args A list of symbols, similar to Lisp lambda lists. `&optional` arguments can be defaulted as in Lisp. Note that, unlike Lisp, `&rest` arguments can also be defaulted. `&rest` arguments can be used in "tail" fashion, as in: `[foo A B . ?quux]`, which matches all `foo` predicates with arguments `A` and `B`, followed by anything else. `&key`, `&aux`, and other lambda-list keywords are not supported.

model-and-other-components
Lists optional models defined with **joshua:define-predicate-model**. You can also use any flavor, as long as it doesn't use **:ordered-instance-variables**. The rules of procedure are identical to those of **defflavor**.

options Any option acceptable to **defflavor**. **:constructor** is unlikely to be useful, as **joshua:define-predicate** already uses it. In addition, see **:destructure-into-instance-variables**, below.

There are two ways that you can make the predicate arguments lexically available to methods. For frequent use, specify the option **:destructure-into-instance-variables** in your predicate definition. This keeps the predicate arguments destructured permanently in each predication, taking up more space but providing faster access. For occasional use you can call the macro **joshua:with-statement-destructured**. Since the macro destructures the ar-

guments each time you call it, it is slower, but such predications take up less space. The latter, for example, is usually appropriate for **joshua:say** methods. The former might be more appropriate for inner loops.

Examples:

```
(define-predicate fruit (a-fruit))

(define-predicate bird (bird) (ltms:ltms-predicate-model))

(define-predicate things-to-pack (traveller &rest objects))

(define-predicate gun (range calibre)
    :destructure-into-instance-variables)

(define-predicate has-disease (patient disease &rest symptoms)
    (:destructure-into-instance-variables disease)) ; partial destructuring
```

Related Functions:

joshua:undefine-predicate
joshua:make-predication
joshua:predicationp

Related Flavor:

joshua:predication

See the section "Joshua Predications" in *User's Guide to Basic Joshua*.

joshua:define-predicate-method *(protocol-function flavor* &rest *options) args* &body *body* — *Macro*

joshua:define-predicate-method defines a particular implementation of a protocol function for a model.

protocol-function	A symbol which is the name of a Joshua protocol function.
flavor	A symbol which is the name of a Joshua predicate model.
options	A list of options for the method type. See the special form **defmethod** in *Symbolics Common Lisp Programming Constructs* for more information. Most Joshua predicate methods will not need any *options*.
args	The list of arguments to the method.
body	The Lisp code which implements the method.

Since most of the protocol functions implement themselves as methods, this expands into a **defmethod** most of the time. However, there are two *caveats* that necessitate your using **joshua:define-predicate-method** instead of a bare **defmethod**:

- Some of the protocol functions are *not* methods, so **joshua:define-predicate-method** has to expand into something different in those cases. (For example, some of the methods have to be in your compile environment, before the predicate/model flavors are around.)

- Some of the protocol functions *are* methods, but use different names or argument conventions than those that are user-visible. For example, **joshua:define-predicate-method** may, for efficiency reasons, decide to implement a protocol function with lots of keyword arguments as an internal function with positional arguments.

Examples:

```
(define-predicate-method (tell tell-error-model) (&rest ignore)
  (error "~S was built using TELL-ERROR-MODEL, so you can't TELL it." self))

(define-predicate-method (say hacker) (&optional (stream *standard-output*))
  (with-statement-destructured (name)
    (format stream "~&~S is a hacker." name)))
```

Related function:

joshua:undefine-predicate-method

See the special form **defmethod** in *Symbolics Common Lisp Programming Constructs*.

joshua:define-predicate-model *name instance-variables components* &body *options* *Macro*

Defines a flavor which may be used as a model or model component for predicates.

name	A symbol which is the name of the model being defined.
instance-variables	A list of the names of instance variables which will keep some of the state of the predicate.
components	A list of component flavors or models.
options	Options to be passed on to **defflavor**.

joshua:define-predicate-model is quite similar to **defflavor**. **joshua:define-predicate-model** forces *name* to be an abstract flavor, and requires **joshua:predication** be a flavor component of any instantiable flavor built on *name*.

Related function:

joshua:undefine-predicate-model

joshua:defquestion *name (control-structure* &rest *control-structure-args) pattern* &key *:code* *Macro*

Defines a question.

name The name of the question.

control-structure Specifies the direction of chaining the question responds to. Currently, only **:backward** chaining questions are supported.

control-structure-args
Like **joshua:defrule**, these are arguments to the control structure. Currently supported are **:importance** and **:documentation**. Both work as they do in rules: The former lets you specify the priority in which you want your questions to run (however, they'll always run after rules); the latter lets you add a string to document the meaning of the question. This string can then be retrieved with the Lisp function **joshua::documentation**.

pattern A single predication. The question triggers when this pattern is matched in an **joshua:ask**, for **:backward** question.

Keywords:

:code Any Lisp code. This is for customized versions of **joshua:defquestion**.

Backward questions behave like backward chaining rules, except that they run *after* all backward rules. They treat the user as an extension of the database, and solicit more solutions from him. (For the basics of rule operation: See the section "Rules and Inference" in *User's Guide to Basic Joshua*.)

Like rules, questions have a name, a trigger pattern, and a body. Like rules, questions are a way of generating information.

When you **joshua:ask** something with **:do-questions joshua::t** and the query pattern unifies with *pattern* in the question, the question body runs. Questions run only after the database has been searched and all appropriate backward rules have been triggered.

If you don't supply the **:code** keyword, **joshua:defquestion** supplies a body for you.

At run time, the query unifies with the question trigger. If there are no logic no logic variables in the unified query, a Yes or No question is generated once. The default answer is No. Answering Yes makes the query that triggered the question succeed. Answering No makes the query fail, which can mean either that the query is known to be **joshua:*false***, or that it is not known to be **joshua:*true***.

If the unified query contains logic variables, the question loops, presenting iterations of an AVV (Accept Variable Values) menu, each soliciting bindings for those variables.

Questions can be used to interact with a user, with some other process running on the machine, or even some other device. For example, a question could go out over the network and ask some other device to answer a question.

Joshua has a default way of asking questions; you can also write your own.

The default version uses either the default **joshua:say** method to format *pattern* or a user-defined **joshua:say** method if available.

Examples:

We define a predicate and then we define a question that triggers on a predication pattern built from this predicate.

```
(define-predicate foo (something something-else))

(defquestion question1 (:backward :documentation "This has no apparent use")
        [foo 1 ?x])
```

Example 1 is a query with no logic variables in the unified query pattern.

```
Example 1:
  (ask [foo 1 2] #'print-query :do-questions t)
Is it true that "[FOO 1 2]"?  [default No]: Yes
[FOO 1 2]
NIL
```

For example 2 we define a **joshua:say** method, and the question uses that method.

```
Example 2:
(define-predicate-method (say foo) (&optional (stream *standard-output*))
  (with-statement-destructured (something something-else) ()
    (format stream "the arguments ~A and ~A are correct"
      something something-else)))

  (ask [foo 1 2] #'print-query :do-questions t)
Is it true that "the arguments 1 and 2 are correct"?  [default No]: Yes
[FOO 1 2]
NIL
```

Example 3 uses a query with logic variables in the query pattern.

Example 3:

```
  (ask [foo 1 ≡z] #'print-query :do-questions t)
For what values of ≡X is it true that "the arguments 1 and ≡X are correct"?
Some solution exists: Yes  No
Value for ≡X: 2
<ABORT> aborts, <END> uses these values

[FOO 1 2]
What are some more values of ≡X such that "the arguments 1 and ≡X are correct"?
Some solution exists: Yes  No
Value for ≡X: BAR
<ABORT> aborts, <END> uses these values

[FOO 1 BAR]
What are some more values of ≡X such that "the arguments 1 and ≡X are correct"?
Some solution exists: Yes  No
<ABORT> aborts, <END> uses these values
NIL
```

To write your own code to do questions, use the **:code** keyword. This keyword takes arguments and a body, as follows:

args (query truth-value continuation &optional query-context)

body The *body* of a **joshua:defquestion** works like Lisp code in the body of a backward rule. If the value of *body* is **nil**, the query that triggered the question fails. If the value of *body* is non-**nil**, the query succeeds. Calling the **joshua:succeed** function explicitly within the *body* allows the query to succeed many times.

Within *body*, *query* is the query predication given to **joshua:ask**, after the query has been unified with the question's trigger.

If *truth-value* is **joshua:*true***, Joshua is trying to determine whether the query is known to be true, as opposed to false or unknown. Similarly for a *truth-value* of **joshua:*false*** Joshua tries to determine whether the query is known to be false, as opposed to true or unknown.

The *query-context* argument can almost always be ignored.

body should do the following:

- If there are no logic variables in the query, decide somehow (perhaps by asking the user a question) if the query is true. If so, call *continuation*. You usually rely on the form **(joshua:succeed)** to call *continuation* for you.

- If there are logic variables present, solicit sets of bindings for them from somewhere (for example, the user). For each such set, call *continuation* (usually via **(joshua:succeed)**).

Examples of custom-written questions:

First we define the predicates, a **joshua:say** method, a question, and a backward rule.

```
(define-predicate wrote (author book))
(define-predicate understands (reader book))

(define-predicate-method (say understands)
                         (&optional (stream *standard-output*))
  (with-statement-destructured (reader book) self
    (format stream "~A understands ~A." reader book)))
```

```
(defquestion writings-of-caesar (:backward) [wrote caesar ?book]
    :code
    ((query truth-value continuation &optional ignore)
     (unless (eql truth-value *true*
                  (error "I can only ask positive questions.")))
     (typecase ?book
       (unbound-logic-variable
         ;;asked with ?book unbound
         (loop for prompt = "Tell me something that Caesar wrote: "
                          then "Tell me something else Caesar wrote: "
               for answer = (accept
                              '((token-or-type (("No more" . no-more))
                                               ((string))))
                              :prompt prompt :default "De Bello Gallico")
               until (eq answer 'no-more)
               do (with-unification
                    (unify ?book answer)
                    (succeed))))
       (otherwise
         ;;asked with ?book bound
         (yes-or-no-p "~&Did Caesar write ~A? " ?book)))))
```

```
(defrule writers-understand-their-work (:backward)
  if [wrote ?author ?work]
  then [understands ?author ?work])
```

Now we **joshua:ask** the query.

```
  (ask [understands Caesar ?book] #'say-query :do-questions t)
Tell me something that Caesar wrote:  [default "De Bello Gallico"]:
 De Bello Gallico
CAESAR understands De Bello Gallico.
Tell me something else Caesar wrote:  [default "De Bello Gallico"]:
 A Canticle for Leibowitz
CAESAR understands A Canticle for Leibowitz.
Tell me something else Caesar wrote:  [default "De Bello Gallico"]: No more
NIL
```

```
  (ask [understands Caesar "Passion on the Nile"] #'say-query :do-questions t)
Did Caesar write Passion on the Nile? (Yes or No) Yes
CAESAR understands Passion on the Nile.
NIL
```

Related Functions:

joshua:undefquestion
joshua:ask
joshua:ask-questions
joshua:map-over-backward-question-triggers
joshua:locate-backward-question-trigger

See the section "Asking the User Questions" in *User's Guide to Basic Joshua*.

joshua:defrule *rule-name (control-structure* &rest *control-structure-args) if if-part then then-part* *Function*

Defines a forward or backward chaining rule. The *control-structure* argument specifies the direction of the rule.

Forward chaining rules respond to new facts entered with **joshua:tell**; the response (that is, the rule body or *then*-part), can involve deducing additional facts that are automatically added to the database, or it can involve executing any Lisp program.

Backward chaining rules respond to a goal entered with **joshua:ask** by trying to satisfy it; this can involve satisfying a series of successive subgoals, or any Lisp program. Backward chaining does not automatically add new facts to the database. See the section "Rules and Inference" in *User's Guide to Basic Joshua*.

rule-name Any symbol that uniquely identifies the rule.

control-structure One of the keywords **:forward** or **:backward** corresponding, respectively, to a forward rule or a backward rule. Future releases may add more possible control structures.

control-structure-args **:importance** lets you control the order of rule execution. **:documentation** lets you add a string that documents the meaning of the rule. Future releases may add more keywords.

:importance takes a *value* argument that can be:

- Numeric; any non-complex number, including +1e∞ or -1e∞ (infinity).

- A symbol (in which case, the system treats it as a special variable whose runtime value should be a number).

- A form; the compiler enwraps it with (lambda () ...) and compiles it. It should return a number when called.

The larger the *value* argument, the higher the priority. Rules with no *value* argument run first, after which rules with a *value* argument are run in order from the highest to the lowest *value*.

Some expense is associated with ordering using **:importance**. In forward chaining rules it causes a "best-first" search through a heap of rules according to the value associated with **:importance**. Backward chaining only orders the local "best-first" search of rules at the current choice point.

A more symbolic type of reasoning, or some level of modeling are usually preferable to the indiscriminate use of **:importance**.

if
The symbol **joshua::if**.

if-part
Specifies the conditions under which the rule succeeds. The *form* of the *if*-part is identical for forward and backward rules. *Procedurally*, the *if*-parts differ depending on rule type:

In *forward* rules the *if*-part is the *trigger* part. It can be one or more predications, joined by **joshua::and** or **joshua::or**. Lisp forms (called *procedural nodes*) can be included in the *if*-part of forward rules, as well. See the section "The Joshua Rule Compiler", page 26.

In *backward* rules the *if*-part is the *action* part. It can be one or more predications as above, as well as any Lisp construct. These become *subgoals*.

then
The symbol **joshua::then**.

then-part
Specifies the conclusions drawn from the rule. The *form* of the *then*-part is identical in forward and backward rules. *Procedurally*, the *then*-parts differ depending on rule type:

In *forward* rules the *then*-part is the *action* part. Can be one or more predications, joined by **joshua::and** or **joshua::or**, as well as any Lisp construct.

In *backward* rules this is the *trigger* part. Must be a single (not a compound) predication.

Note that the *if* and *then* clauses can occur in either order. For example, some programmers prefer to place the *then*-part of backward rules first, so that the trigger (procedure head) always comes first. Either of the arrangements shown below is valid.

```
If [...] Then [...]
```

and

```
Then [...] If [...]
```

A rule's action part (the *then*-part of forward rules, and the *if*-part of backward rules) can specify any suitable action(s), such as adding or retracting predications, using Lisp code to perform embedded tests or computations, calling **joshua:ask** or **joshua:tell**, interacting with the user, or displaying messages. When your Lisp code does iterations, call the function **joshua:succeed** inside it to let Joshua know that the current set of bindings is correct. Otherwise, Lisp code "succeeds" by returning non-**nil**. See examples below.

If the action part of a forward rule contains a predication that is not embedded in Lisp code, this newly deduced fact is automatically added to the database when the rule executes (a **joshua:tell** is implicit). Note that such a predication can be backquoted. If the predication is embedded in Lisp, however, you must explicitly use a **joshua:tell** to insert the fact into the database.

The action part of a backward rule has an implicit **joshua:ask** around it. Backward rule action parts add no predications to the database, unless you explicitly use a **joshua:tell** to accomplish this.

A backward rule's trigger part (the *then*-part) must consist of a single predication. The trigger can contain logic variables. These variables are bound by the unifier when the trigger part of the rule is matched against the query; they are then passed to the action part (the *if*-part).

A forward rule's trigger part (the *if*-part) may contain an arbitary number of predications and Lisp forms. The triggers can contain logic variables. A forward rule's triggers behave as follow:

- If the trigger is a predication, it is *satisfied* when it has been matched against a predication in the database. The logic variables in the trigger are bound by the unifier when the trigger part of the rule is matched against the database predication.

- The trigger may be a Lisp form (we call such triggers *procedural triggers*). Such a trigger may be satisfied in two ways: If it returns **joshua::t**, it is regarded as satisfied. It is also regarded as satisfied each time it calls **joshua:succeed**.

- If a procedural trigger never calls **joshua:succeed**, but merely returns **joshua::t** or **joshua::nil**, then it acts as a *filter* on the previous triggers (either accepting or rejecting the bindings produced by its predecessors).

- A procedural trigger may also act as a *generator*, producing several acceptable sets of bindings and calling **joshua:succeed** for each one.

- Logic variables which occur for the first time in a procedural trigger may be bound by calling **joshua:unify**. Logic variables that are referenced in a procedural trigger but which occur in an earlier trigger, are bound to the value established by the earlier trigger during the execution of the Lisp trigger.

- The logical connective *and* can be used to group the triggers into subgroups all of which must be satisfied. The logical connective *or* can be used to group the patterns into subgroups any one of which must be satisfied.

- The trigger part of a forward rule can include the keyword **:support** followed by a logic variable after any trigger pattern. During the execution of the rule, this logic variable is bound to the predication that matched the trigger pattern immediately preceding the keyword **:support**.

- A procedural trigger may provide an argument to **joshua:succeed** which should be a *database-predication*. If it does so, this predication is treated as if it had matched a normal trigger of the rule. If there is a **:support** keyword following the procedural trigger, then the logic variable following it will be bound to the *database-predication*.

Joshua stores and retrieves rules by their triggers. When a new rule is defined, the rule compiler stores the rule's trigger in a place appropriate to the rule type. The system finds and executes applicable rules by locating their triggers; similarly, it deletes unwanted rules by removing their triggers. See the section "The Joshua Rule Indexing Protocol", page 36.

Here are some examples. First, here's how to use the **:documentation** keyword. We use a forward rule as an example, but **:documentation** works identically for backward rules.

```
(define-predicate reads (person how-much))
(define-predicate is-bookworm (person))

(defrule simple (:forward :documentation "Identifies bookworms")
  if [reads ?person constantly]
  then [is-bookworm ?person])
```

To retrieve the documentation string of this rule, use the Lisp function **joshua::documentation**.

```
(documentation 'simple)
"Identifies bookworms"
```

Here are some examples of forward chaining. This first a simple declarative rule:

```
(defrule good-cake (:forward)
  if [and [rises ?cake justright]
          [color ?cake evenly-gold]
          [texture ?cake moist]
          [taste ?cake justright]]
  then [good ?cake])
```

Next is an example of using the :support keyword to allow the body of the rule to reference the triggering facts:

```
(defrule good-cake (:forward)
  if [and [rises ?cake justright] :support ?f1
          [color ?cake evenly-gold] :support ?f2
          [texture ?cake moist] :support ?f3
          [taste ?cake justright] :support ?f4
          ]
  then [and (Format t "~%The reason I thing that ~s is good is that:"
                        ?cake)
              (say ?f1) (say ?f2) (say ?f3) (say ?f4)
              [good ?cake]])
```

Here we show how a Procedural Trigger can be used as a generator. Once all triggers before the procedural trigger are matched, it executes and generates two acceptable bindings for ?color.

```
(defrule good-cake (:forward)
  if [and [rises ?cake justright]
          [texture ?cake moist]
          (loop for color in '(evenly-gold nicely-brown)
                do (unify ?color color)
                   (succeed))
          [taste ?cake justright]
          ]
  then [and (format t "~&~s is a good cake with color ~s"
                       ?cake ?color)
              [good ?cake]])
```

Here is an example of a procedural trigger being used as a filter:

```
(defrule check-temperature (:forward)
  if [and [temperature-used ?object ?temp]
          (< 325 ?temp 400)]  ; example of Lisp used as a filter
  then [correct-temperature-used ?object ?temp])
```

```
(defun check-oven-setting ()
  (clear)
  (tell [temperature-used jelly-roll 375])
  (ask [correct-temperature-used jelly-roll ?temp] #'print-query))

(check-oven-setting)
[CORRECT-TEMPERATURE-USED JELLY-ROLL 375]
NIL
```

Finally, here is an example using nested *and*'s and *or*'s:

```
(defrule deduce-ancestry (:forward)
  if [or [is-parent-of ?old ?young]
         [and [is-ancestor-of ?old ?middle]
              [is-parent-of ?middle ?young]]]
  then [is-ancestor-of ?old ?young])
```

Here are some examples using backward chaining:

```
(defrule sailor-alert (:backward)
  if [or [condition-of wind gusting]
         [weather-forecast squalls]]
  then [issue-warning small-craft alert])

;;; Lisp code in action part of backward rule
(define-predicate good-to-read (book))
(defparameter *books* '(decameron canterbury-tales gargantua-and-pantagruel
                                  tom-jones catch-22))

(defrule reading-list (:backward)
  if (typecase ?candidate-book
       (unbound-logic-variable
        (loop for book in *books*
               doing (with-unification
                       (unify ?candidate-book book)
                       (succeed))))
       (otherwise
        (member ?candidate-book *books*)))
  then [good-to-read ?candidate-book])

(ask [good-to-read ?x] #'print-query)
[GOOD-TO-READ DECAMERON]
[GOOD-TO-READ CANTERBURY-TALES]
[GOOD-TO-READ GARGANTUA-AND-PANTAGRUEL]
[GOOD-TO-READ TOM-JONES]
[GOOD-TO-READ CATCH-22]
NIL
```

You can inhibit backward chaining rule invocation by passing **joshua::nil** as the **:do-backward-rules** keyword argument to **joshua:ask** (the default value is **joshua::t**). In this case the system does only a database lookup.

You can cause backward question invocation by passing **joshua::t** as the **:do-questions** keyword argument to **joshua:ask** (the default is **joshua::nil**).

Advanced Concepts Note:

Six built-in models are available for predicates in **joshua:ask** goals. These flavors do subsets of what **joshua:ask** normally does, by leaving out one or more of the steps **joshua:ask-data**, **joshua:ask-rules**, or **joshua:ask-questions**. Thus the models save a certain amount of overhead when their predicates are used as goals to **joshua:ask**. The steps that *are* done are indicated by the names:

- **joshua:ask-data-only-mixin**
- **joshua:ask-rules-only-mixin**
- **joshua:ask-questions-only-mixin**
- **joshua:ask-data-and-rules-only-mixin**
- **joshua:ask-data-and-questions-only-mixin**
- **joshua:ask-rules-and-questions-only-mixin**

Related Functions:

joshua:undefrule
joshua:tell
joshua:ask
joshua:ask-rules

See the section "Rules and Inference" in *User's Guide to Basic Joshua*. See the section "The Joshua Rule Facilities ", page 23.

joshua:delete-backward-question-trigger *predication truth-value question-name context* *Generic Function*

predication	The pattern under which the backward question is indexed.
truth-value	The truth value of the pattern under which the backward question is indexed.
question-name	The name of the question to be deleted.
context	The entire trigger part of the backward question. Useful in advanced modeling applications.

joshua:undefquestion calls this protocol function with the pattern from the trigger part of a backward question. The function "unindexes" the trigger data-structure of the backward question which corresponds to the pattern. After the pattern is "unindexed" the question is no longer accessible.

Tailoring of backward-question indexing is usually accomplished by providing methods for the **joshua:locate-backward-question-trigger** and **joshua:map-over-backward-question-triggers** protocol functions. The

joshua:add-backward-question-trigger and **joshua:delete-backward-question-trigger** methods provided as Joshua's defaults call **joshua:locate-backward-question-trigger** as a subroutine. All of the interesting tailoring of their behavior can be obtained by providing a **joshua:locate-backward-question-trigger** method.

However, it might be useful in some applications to provide **:before** or **:after** methods for the add and delete methods, for example to keep track of what questions are currently indexed where. Even in advanced modeling applications it is unlikely that you will need to define a method for the **joshua:delete-backward-question-trigger** protocol function.

See the section "The Joshua Question Indexing Protocol", page 48.

joshua:delete-backward-rule-trigger *predication truth-value rule-name context* *Generic Function*

predication	The pattern under which a backward rule is indexed.
truth-value	The truth value of the pattern under which the backward rule is indexed.
rule-name	The name of the rule to be deleted.
context	The entire *if*-part of the rule. Useful for advanced modeling.

joshua:undefrule calls this protocol function with the pattern from the *then-* part of a backward chaining rule. The function "unindexes" the trigger data-structure of the backward rule which corresponds to the pattern. After the pattern is "unindexed" the rule is no longer accessible. Tailoring of backward rule indexing is usually accomplished by providing methods for the **joshua:locate-backward-rule-trigger** and **joshua:map-over-backward-rule-triggers** protocol functions. The **joshua:add-backward-rule-trigger** and **joshua:delete-backward-rule-trigger** methods provided as Joshua's defaults call **joshua:locate-backward-rule-trigger** as a subroutine. All of the interesting tailoring of their behavior can be obtained by providing a **joshua:locate-backward-rule-trigger** method.

However, it might be useful in some applications to provide **:before** or **:after** methods for the add and delete methods, for example to keep track of what rules are currently indexed where. Even in advanced modeling applications it is unlikely that you will need to define a method for the **joshua:delete-backward-rule-trigger** protocol function.

See the section "The Joshua Rule Indexing Protocol", page 36.

joshua:delete-forward-rule-trigger *predication truth-value rule-name context* *Generic Function*

predication	The pattern under which the forward rule is indexed.

truth-value	The truth value of the pattern under which the forward rule is indexed.
rule-name	The name of the rule to be deleted.
context	The entire *if*-part of the rule. Useful for advanced modeling.

joshua:undefrule calls this protocol function once for each pattern in the If part of a forward chaining rule. The function "unindexes" the trigger data-structure of the forward rule which corresponds to the pattern. After each pattern is "unindexed" the rule is no longer accessible. Tailoring of forward rule indexing is usually accomplished by providing methods for the **joshua:locate-forward-rule-trigger** and **joshua:map-over-forward-rule-triggers** protocol functions. The **joshua:add-forward-rule-trigger** and **joshua:delete-forward-rule-trigger** methods provided as Joshua's defaults call **joshua:locate-forward-rule-trigger** as a subroutine. All of the interesting tailoring of their behavior can be obtained by providing a **joshua:locate-forward-rule-trigger** method.

However, it might be useful in some applications to provide **:before** or **:after** methods for the add and delete methods, for example to keep track of what rules are currently indexed where. Even in advanced modeling applications it is unlikely that you will need to define a method for the **joshua:delete-forward-rule-trigger** protocol function.

See the section "The Joshua Rule Indexing Protocol", page 36.

joshua:different-objects *object1 object2* *Function*

Returns **nil** if the arguments are **eql** or if either argument is an uninstantiated logic variable (in the latter case the two objects can potentially be *made* to be the same by the unifier). Otherwise, **joshua:different-objects** returns **t**.

object1	A Lisp object.
object2	A Lisp object.

This function is useful in making rules that weed out inappropriate self-referential behavior. For example, in a program simulating the behavior of a monkey that can pick up objects, it is important to ensure that the monkey does not try to pick up itself.

This function is often useful in the *if*-part of rules, or in Lisp code.

```
(defrule pick-up (:backward)
  if (different-objects ?obj 'monkey)
  then [can-pick-up monkey ?obj])
```

To invoke this rule, you would type something like:

```
(ask [can-pick-up monkey wrench] #'print-query)
```

See the section "Using Joshua Within Lisp Code" in *User's Guide to Basic Joshua*.

Disable Joshua Tracing Command

Turns off Joshua tracing.

Type of Tracing The type of tracing to disable. It can be one of forward rules, backward rules, predications, TMS operations, or all. The type-of tracing defaults to all.

Disable Joshua Tracing turns off the Joshua tracing facility.

Related Commands:

"Enable Joshua Tracing Command"
"Reset Joshua Tracing Command"

joshua:discrimination-net-clear *root-node* *Function*

joshua:discrimination-net-clear clears all the data out of the discrimination net whose root is *root-node*. This function works by lopping off all the outgoing arcs from *root-node*. The garbage collector reclaims all the descendants.

joshua:discrimination-net-clear is called by **(joshua:clear joshua:discrimination-net-data-mixin)**.

root-node The root node of a discrimination net.

Related Functions:

joshua:discrimination-net-uninsert
joshua:discrimination-net-insert

See the section "The Joshua Database Protocol", page 8.

joshua:discrimination-net-data-mixin *Flavor*

This flavor provides the Joshua default behavior for storing data predications in a discrimination net. It has methods for **joshua:fetch**, **joshua:insert**, **joshua:uninsert**, and **joshua:clear**.

Related topics:

joshua:default-predicate-model
joshua:discrimination-net-fetch
joshua:discrimination-net-insert
joshua:discrimination-net-uninsert
joshua:discrimination-net-clear

joshua:discrimination-net-fetch *root-node predication continuation root-node predication continuation* *Function*

joshua:discrimination-net-fetch searches the discrimination net whose root is *root-node* using *predication* as a pattern, and calls *continuation* for each item in the discrimination net that might unify with *predication*.

joshua:discrimination-net-fetch is called by **(joshua:fetch joshua:discrimination-net-data-mixin)**. It is the default implementation of the **joshua:fetch** generic function for the virtual database. **joshua:fetch** does the data retrieval for the Joshua protocol function **joshua:ask**, which satisfies backward goals.

root-node	The root node of a discrimination net.
predication	A predication to be searched for.
continuation	A function of one argument, to be called on each candidate the discrimination net finds.

Related Functions:

joshua:discrimination-net-insert
joshua:discrimination-net-clear

See the section "The Joshua Database Protocol", page 8.

joshua:discrimination-net-insert *root-node predication* *Function*

joshua:discrimination-net-insert takes *predication* and inserts it into the discrimination net whose root is *root-node*. It is called by **(joshua:insert joshua:discrimination-net-data-mixin)**, the default implementation of the **joshua:insert** generic function for the virtual database. **joshua:insert** is the first step of the Joshua protocol function **joshua:tell**, that adds data into the database.

joshua:discrimination-net-insert adds data to the discrimination net by side-effecting a leaf node (that is, adding a predication). The appropriate nodes in the discrimination net are created if necessary.

root-node	The root node of a discrimination net.
predication	is a predication to be added to the database.

The discrimination net discriminates fully with two exceptions. Logic variable arguments are not uniquely identified; they discriminate to a node labeled **ji::*variable***. Similarly, embedded lists discriminate to a node labeled **ji::*embedded-list***. That is, as far as the discrimination net is concerned, all variables are alike, and all lists are alike.

joshua:discrimination-net-insert does not deal with any justification, forward-rule triggering, or unification issues.

joshua:discrimination-net-insert returns two values:

- The canonical version of *predication* that was stored in the database. If another predication that is a variant of *predication* already exists in the database, **joshua:discrimination-net-insert** returns the older version. See the function **joshua:variant**, page 252.

- A flag that determines whether *predication* was added to the database or not. This flag is either **t** if *predication* is newly added, or **nil** if a variant was already in the database. Note that this is what **joshua:insert** is contracted to return; thus, **joshua:discrimination-net-insert** is one possible realization of **joshua:insert**.

Related Functions:

joshua:discrimination-net-uninsert
joshua:discrimination-net-fetch
joshua:discrimination-net-clear

See the section "The Joshua Database Protocol", page 8.

joshua:discrimination-net-uninsert *root-node predication* *Function*

This is the dual to **joshua:discrimination-net-insert**. It removes *predication* from the discrimination net whose root node is *root-node*. For example, this is called by **(joshua:uninsert joshua:discrimination-net-data-mixin)** to implement the **joshua:uninsert** generic function for the default data model.

root-node The root of a discrimination net.

predication The database predication to be removed from the discrimination net. This must be the actual predication object from the database, and not a copy.

Related Functions:

joshua:discrimination-net-insert
joshua:discrimination-net-clear

See the section "The Joshua Database Protocol", page 8.

Enable Joshua Tracing Command

Turns on Joshua Tracing.

Type of Tracing The type of tracing to enable. You can enable the tracing of forward rules, backward rules, predications, TMS operations, or All. Unless otherwise specified (by using the :*Menu* option for example), tracing is turned on with the same options and tracing events that were in effect the last time you used tracing.

:*Menu* Brings up a menu of detailed tracing options for the *type of tracing* being enabled. This menu provides a greater degree of control over exactly what gets traced and when the tracing facility interacts with the user.

:*Trace Events* When enabling a particular type of tracing this option allows you to specify precisely which events will be displayed during tracing. These can also be set by using the :*Menu* option.

:*Step Events* Allows you to specify at which events the tracing facility will stop and prompt for interaction. These can also be set by using the :*Menu* option.

The Enable Joshua Tracing command turns on the Joshua tracing tools and allows you to customize tracing to your particular application or preference. The Joshua tracing facility is very flexible. You can, for example, trace just forward rules that are triggered by predications matching a particular pattern:

```
Enable Joshua Tracing Forward Rules :Menu Yes
```

```
Forward Rules

Forward Rules Options
Trace forward rules: All  Selectively
Trace forward rules: None  forward rules
Trace forward rule triggers: None  [TME:LOVES DEMETRIUS HERMIA]
Traced Events : Fire  Exit  Queue  Dequeue
Stepped Events: Fire  Exit  Queue  Dequeue
<ABORT> aborts, <END> uses these values
```

Or, you can even just trace predications built on a particular model:

```
Enable Joshua Tracing Predications :Menu Yes
```

```
Predications

Predications Options
Trace Predications: All  Selectively
Trace Predicates of flavor(s): None  LTMS:LTMS-PREDICATE-MODEL
Trace Facts Matching: None  predications
Traced Events : Ask  Tell  Untell  Truth value change  Justify  Unjustify
Stepped Events: Ask  Tell  Untell  Truth value change  Justify  Unjustify
```

The best way to familiarize yourself with this facility is to type Enable Joshua Tracing All :Menu Yes. This brings up a menu of all the types of Joshua tracing and the options available for each one. By moving the mouse over each option you can see the documentation for that option in the mouse documentation line.

Related Commands:

"Disable Joshua Tracing Command"
"Reset Joshua Tracing Command"

See the section "Tracing Predications" in *User's Guide to Basic Joshua*. See the section "Tracing Rules" in *User's Guide to Basic Joshua*.

joshua:equated *slot1 slot2* *Joshua Predicate*

This predicate is part of the Joshua object facility. It is used to assert and query the equality-links between slots of Joshua objects.

Note that where equalities are between attributes of different sub-parts of the same object, and when those equalities hold for all objects of a certain type, it may be easier to declare those equalities at the time when the class of objects is defined by **joshua:define-object-type**.

ltms:equated *slot1 slot2* *Joshua Predicate*

This predicate is part of the Joshua object facility. It is used in the same manner as **joshua:equated**, except it refers to slots whose values are truth-maintained. Slots are declared as truth-maintained at the time the class of objects is defined by **joshua:define-object-type**.

joshua:equated-mixin *Flavor*

This flavor-mixin is part of the Joshua object facility. It may be used to add equality-link behaviour, like that of the default equality predicate **joshua:equated**, to predicate models defined by the user.

joshua:expand-forward-rule-trigger *rule-trigger support-variable-name truth-value context* *Generic Function*

predication A trigger-pattern of a forward chaining rule.

support-variable-name
The name (if any) of the logic-variable which should be bound to the object which matches the pattern.

truth-value The truth value of the pattern.

context The entire If-part of the rule. This can be useful in advanced modelling applications.

joshua:expand-forward-rule-trigger is called by the Joshua rule compiler as the first step of translating the syntax of a forward-chaining rule into compiled Lisp code.

joshua:expand-forward-rule-trigger is called once for each predication included in the trigger of the rule. Its job is to return a list structure that explains to the rule compiler how to process the pattern.

For example in the following rule:

```
(defrule foobar (:forward)
  If [and [foo1 ?x ?y] :support ?f1
          [not [foo2 ?y ?z]] :support ?f2
          ]
  Then [foo3 ?x ?y ?z])
```

joshua:expand-forward-rule-trigger will be called three times (once for the entire **joshua::and** and then once for each predication inside the **joshua::and**). **joshua:expand-forward-rule-trigger** takes four arguments: the pattern to expand, the name of its :support variable (or nil), its truth-value and the entire If-part (which can be treated as the "context" of the pattern). Thus, the arguments passed in for these three calls wil be:

```
[and [foo1 ?x ?y] :support ?f1
     [not [foo2 ?y ?z]] :support ?f2] nil *true* and <the whole If-part>
[foo1 ?x ?y] ?f1 and *true* <the whole If-part>
[foo2 ?y ?z] ?f2 *false* <the whole If-part>
```

Note that although we have displayed the patterns as if they were predications, this is not actually true. **joshua:expand-forward-rule-trigger** runs at compile time and manipulates a source-code representation of predications and logic-variables, see the section "The Source Representaton of Predications and Logic-variables".

joshua:expand-forward-rule-trigger should return a list structure (called a *trigger-description*) which must be one of the following forms:

1. `(:match` *pattern name truth-value*`)`. This trigger description informs the rule compiler that the current trigger should be treated simply as a pattern to be matched.

   - *pattern* is the predication that represents the pattern to be matched.
   - *name* is the logic variable which the rule triggering mechanisms should bind to the predication that matched this trigger.
   - *truth-value* (which in the current implementation should be either **joshua:*true*** or **joshua:*false***) is the truth value which the matching predication is required to have in order to trigger the rule.

2. `(:and` *trigger-descriptions*`)` This trigger description informs the rule compiler that the current pattern is actually a conjunction of patterns all of which must be matched to trigger the rule. The system-provided default method for AND predications returns this type of trigger description. The second element of the trigger description must be a list of trigger descriptions, i.e. lists returned by calling **joshua:expand-forward-rule-trigger**.

3. `(:or` *trigger-descriptions*`)` This trigger description informs the rule compiler that the current pattern is actually a disjunction of patterns any of which must be matched to trigger the rule. The system provided default method for OR predications returns this type of trigger description. The second element of the trigger description must be a list of trigger descriptions, i.e. lists returned by calling **joshua:expand-forward-rule-trigger**.

4. `(:procedure` *lisp-expression name*`)` This trigger description informs the rule compiler that the current trigger is not a pattern to be matched, but rather a Lisp expression that appears in the trigger position. Such expressions are executed once all proceeding patterns in the rule have been matched. The expression can act as a filter by returning either **joshua::t** or **joshua::nil**. **joshua::t** indicates success; in this case the bindings accumulated up to this point are considered acceptable and rule triggering continues. **joshua::nil** indicates failure; in this case the bindings are considered unacceptable.

The expression can also act as a generator in which it produces several new sets of bindings each of which is consistent with the bindings that were in effect when the rule was triggered. To do this it should bind whatever logic-variables it wants to and then call **joshua:succeed**. **joshua:succeed** takes a rest-argument; the rule compiler will arrange for this values passed to **joshua:succeed** to be bound to the logic-variable which is the third element of the trigger description.

See the function **joshua:succeed**, page 232.

5. `(:ignore)` This trigger description informs the rule compiler that it should ignore this trigger. The are two reasons for using this type of trigger description. The first is to allow a rule to have patterns included in it simply for the sake of clarity. The second is to include patterns only to specify context.

A Procedural trigger description can be used to implement a mixed-chaining strategy in which a forward-rule trigger invokes backward chaining capabilities. This would be useful if it is known that a particular type of predication is never actually asserted but is only deduced by backward chaining rules.

The following rule is how one would implement this mixed-chaining strategy if it were known that `FOO2` predications are only deduced by backward chaining rules:

```
(define-predicate foo1 (a b))
(define-predicate foo2 (a b))
(define-predicate backward-foo2 (a b))
(define-predicate foo3 (a b c))

(defrule foo (:forward)
  If [and [foo1 ?a ?b]
          (ask [foo2 ?b ?c]
               #'(lambda (ignore) (succeed)))]
  Then [foo3 ?a ?b ?c])

(defrule foo2-backward (:backward)
  If [backward-foo2 ?b ?a]
  Then [foo2 ?a ?b])
```

The structure of the rete network for this rule is a simple linear chain consisting of a match node followed by a procedural node (acting as a generator) as shown in figure 30.

If we execute the following two **joshua:tell**'s then the rule will be triggered by the second statement which matches the first pattern of the rule. Execution then proceeds to the procedural node which chains backward using the rule `FOO2-BACKWARD`. This is shown in figure 31.

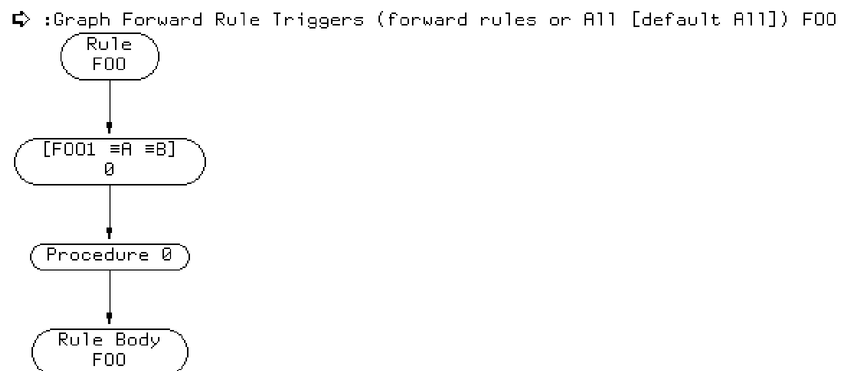


Figure 40. Graph of the Mixed Chaining Rule Foo

```
(tell [backward-foo2 3 2])
 ▶ Telling predication [BACKWARD-FOO2 3 2]
[BACKWARD-FOO2 3 2]
T
➪
(tell [foo1 1 2])
 ▶ Telling predication [FOO1 1 2]
 ▶ Asking predication [FOO2 2 ≡C]
 ▶ Trying backward rule FOO2-BACKWARD  (Goal... )
   ▶ Asking predication [BACKWARD-FOO2 ≡C 2]
 ▶ Succeeding backward rule FOO2-BACKWARD
 ▶ Firing forward rule FOO (1 trigger)
   ▶ Telling predication [FOO3 1 2 3]
```

Figure 41. Trace of The Mixed Chaining Rule Foo

However this rule can be made more declarative appearing by using **joshua:expand-forward-rule-trigger** as follows:

```
(define-predicate-model mixed-chaining-mixin () ())

(define-predicate-method
  (expand-forward-rule-trigger mixed-chaining-mixin)
  (name truth-value ignore)
  (let ((query (if (eql truth-value *true*)
                   self
                   `[not ,self])))
    `(:procedure (prog1 nil
                        (ask ,query
                             #'(lambda (ignore)
                                 (succeed))))
      ,name)))
```

```
(define-predicate foo2 (a b)
  (mixed-chaining-mixin default-predicate-model))

(defrule foo (:forward)
  If [and [foo1 ?a ?b]
          [foo2 ?b ?c]]
  Then [foo3 ?a ?b ?c])
(clear)
(tell [backward-foo2 3 2])
(tell [foo1 1 2])
```

Now the rule FOO appears to simply match two patterns. However, it actually compiles into exactly the same rete network as shown in figure 40.

Sometimes using **joshua:ask** in the trigger part of a rule may not be the appropriate way to achieve a mixed chaining strategy. One reason, is that **joshua:ask** queries the world for facts that are deducible at that moment. If a new fact arrives later that would have made the goal deducible, **joshua:ask** will, of course, not notice this. However, forward chaining rules should draw conclusions whenever the data warrants the deduction.

A solution to this problem is to use a more explicit form of reasoning in which goal directed reasoning is conducted by forward rules which are triggered by explicit predications stating the existence of a goal.

Here is an alternative mixed chaining scheme which implements backward chaining by explicitly telling show predications. These trigger forward rules which then work to find a way to satisfy the goal included in the show statement.

For example, the following rule:

```
(defrule foo2-explicit-goal (:forward)
  If [and [show [foo2 ?a ?b]]
          [backward-foo2 ?b ?a]]
  Then [foo2 ?a ?b])
```

Will deduce FOO2 anytime that BACKWARD-FOO2 is asserted and there is a SHOW predication stating that we want this conclusion to be drawn. The rule is more flexible than a backward rule, since it does not depend on the relative order of posting the goal and asserting the data necessary to deduce it. (Of course, this rule is also less efficient than a backward rule).

We can use **joshua:expand-forward-rule-trigger** just as we did in the previous section to make the rule FOO use this form of mixed chaining while retaining its declarative appearance, as follows:

```
(define-predicate-model mix-chain-mixin ()
  ())

(defvar *inside-alternative-backward-chaining-mixin* nil)
```

```
(define-predicate-method
  (expand-forward-rule-trigger mix-chain-mixin)
  (name truth-value context)
  (if *inside-alternative-backward-chaining-mixin*
      `(:match ,self ,name ,truth-value)
      (let ((*inside-alternative-backward-chaining-mixin* t))
        (let ((query (if (eql truth-value *true*)
                         self
                         `[not ,self])))
          `(:and
             ,(expand-forward-rule-trigger
                `(tell [show ,query]) nil *true* context)
             ,(expand-forward-rule-trigger
                self name truth-value context))))))

(define-predicate show (predication))

(define-predicate foo2 (a b)
  (mix-chain-mixin default-predicate-model))
```

This **joshua:expand-forward-rule-trigger** method expands the `FOO2` pattern of the rule into two components. The first **joshua:tell**'s the `SHOW` statement that triggers the `FOO2-EXPLICIT-GOAL` rule. The second is a simple match node that waits for the `FOO2` goal to become true. The **joshua:expand-forward-rule-trigger** method is somewhat tricky because it wants to expand the intial `[foo2 ... ]` pattern into two nodes, one of which **joshua:tell**s `[show [foo2 ... ]]` and the other of which matches `[foo2 ... ]`. A special variable is bound to prevent an infinite recursion in the expansion of this pattern.

Figure32shows the Rete net for this rule.

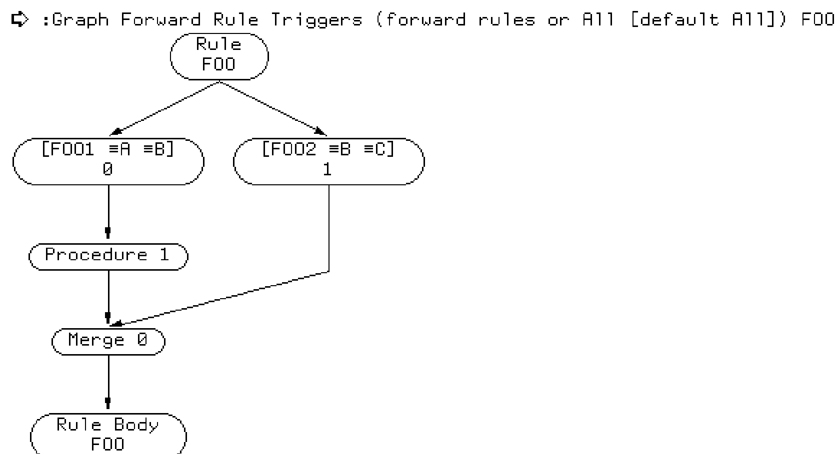


Figure 42. Graph of Mixed Chaining Rule Foo

Notice that the rule contains two match nodes, one for each pattern. The match node for the FOO1 pattern leads to a procedural node which **joshua:tell**s a [show [foo2 ...]] predication and then **joshua:succeed**s. Following this the two paths merge. If the Foo1 statement is asserted first the rule will assert the SHOW statement which will cause the FOO2-EXPLICIT-GOAL rule to wait for a FOO2-BACKWARD statement. At which point the FOO2-EXPLICIT-GOAL rule will assert a FOO2 statement which will match the other trigger pattern of the FOO rule. If the facts are asserted in the other order, the rule will also deduce the desired conclusion, as shown in figures 33 and

```
(tell [backward-foo2 3 2])
 ▶ Telling predication [BACKWARD-FOO2 3 2]
[BACKWARD-FOO2 3 2]
T
➪
(tell [foo1 1 2])
 ▶ Telling predication [FOO1 1 2]
 ▶ Telling predication [SHOW [FOO2 2 ≡C]]
 ▶ Firing forward rule FOO2-EXPLICIT-GOAL (2 triggers)
   ▶ Telling predication [FOO2 2 3]
 ▶ Firing forward rule FOO (2 triggers)
   ▶ Telling predication [FOO3 1 2 3]
[FOO1 1 2]
```

Figure 43. Trace of Explicitly Controlled Mixed Chaining

```
(tell [foo1 1 2])
 ▶ Telling predication [FOO1 1 2]
 ▶ Telling predication [SHOW [FOO2 2 ≡C]]
[FOO1 1 2]
T
➪
(tell [backward-foo2 3 2])
 ▶ Telling predication [BACKWARD-FOO2 3 2]
 ▶ Firing forward rule FOO2-EXPLICIT-GOAL (2 triggers)
   ▶ Telling predication [FOO2 2 3]
   ▶ Firing forward rule FOO (2 triggers)
     ▶ Telling predication [FOO3 1 2 3]
[BACKWARD-FOO2 3 2]
```

Figure 44. Trace of Explicitly Controlled Mixed Chaining

Here's an example using the :ignore trigger description:

```
(defrule adder-forward (:forward)
  If [and [type-of ?a adder]
          [Value-of addend ?a ?value-1]
          [Value-of augend ?a ?value-2]]
  Then `[value-of output ?a ,(+ ?value-1 ?value-2)])
```

A trigger-indexing scheme might be used which guarantees that this rule will only be triggered by Value-of assertions that describe the values of the ADDEND and AUGEND of adders. In such a case the first pattern is required during rule compilation to inform the **joshua:locate-forward-rule-trigger** method that it is indexing patterns having to do with adders. However,

once such a trigger-indexing scheme is established the first pattern is actually redundant.

```
(define-predicate-method
  (expand-forward-rule-trigger type-of-model) (ignore ignore ignore)
  `(:ignore))


(define-predicate type-of (object type)
                (type-of-model default-protocol-implementation-model))
```

See the section "The Joshua Rule Compiler", page 26.

joshua:expand-backward-rule-action *rule-action support-variable-name truth-value other-ask-args context* — *Joshua Protocol Method*

predication — An action of a backward chaining rule (i.e. part of the If-part)

name — The name (if any) of the logic-variable which should be bound to the backward support of this query.

truth-value — The truth value of the pattern.

other-ask-args — Keyword arguments to **joshua:ask** which should be included with this query.

context — The entire If-part of the rule, which can be regarded as the context of this query.

joshua:expand-backward-rule-action is called by the Joshua rule compiler as the first step of translating the syntax of a backward-chaining rule into compiled Lisp code.

What the Backward Rule-compiler Does to the Actions of a Rule

The backward rule compiler turns the If-part of a rule into a series of nested **joshua:ask**'s. For example, the actions of the following rule:

```
(defrule foobar (:backward)
  If [and [foo1 ?x ?y] :support ?f1 :do-backward-rules nil
          [not [foo2 ?y ?z]] :support ?f2
          ]
  Then [foo3 ?x ?y ?z])
```

are converted into a highly optimized version of the following code:

```
(ask [foo1 ?x ?y]
     #'(lambda (support2196)
         (unify ?f1 support2196)
         (ask [not [foo2 ?y ?z]]
              #'(lambda (support2197)
                  (unify ?f2 #:support2197)
                  (let ((ji::rule-support
                          (list ji::.goal. ji::.truth-value.
                                '(rule foobar)
                                support2196 support2197)))
                    (funcall ji::.continuation. ji::rule-support))))
         :do-backward-rules nil))
```

The backward rule compiler also handles the keyword arguments which can be attached to patterns in the If-part of the rule. See the section "Advanced Features of Joshua Rules", page 24.

The Contract of the Generic Function joshua:expand-backward-rule-action

The **joshua:expand-backward-rule-action** protocol function controls how the conversion is performed.

joshua:expand-backward-rule-action is called once for each predication included in the If-part of the rule. Its job is to return a list structure that explains to the rule compiler how to process the pattern.

For example in the following rule:

```
(defrule foobar (:backward)
  If [and [foo1 ?x ?y] :support ?f1 :do-backward-rules nil
          [not [foo2 ?y ?z]] :support ?f2
          ]
  Then [foo3 ?x ?y ?z])
```

joshua:expand-backward-rule-action will be called three times (once for the entire **joshua::and** and then once for each predication inside the **joshua::and**). **joshua:expand-backward-rule-action** takes five arguments: the pattern to expand, the name of its :support variable (or nil), its truth-value, the value of the keyword arguments attached to this pattern that should be passed onto **joshua:ask** (e.g. :do-backward-rules and :do-questions) and the entire If-part (which can be treated as the "context" of the pattern). Thus, the arguments passed in for these three calls wil be:

```
[and [foo1 ?x ?y] :support ?f1 :do-backward-rules nil
     [not [foo2 ?y ?z]] :support ?f2] nil *true* (t t) <the whole If-part>
[foo1 ?x ?y] ?f1 *true* (nil t) <the whole If part>
[foo2 ?y ?z] ?f2 *false* (t t) <the whole If part>
```

Note that although we have displayed the patterns as if they were predications, this is not actually true. **joshua:expand-backward-rule-action** runs at compile time and manipulates a source-code representation of predications and logic-variables, see the section "The Source Representaton of Predications and Logic-variables".

joshua:expand-backward-rule-action should return a list structure (called a *action-description*) which must be one of the following forms:

1. `(:match` *pattern name truth-value* ask-keyword-args`)`. This action description informs the rule compiler that the current action should be treated simply as a pattern to be **joshua:ask**'ed. This action will compile into an **joshua:ask** form whose continuation will perform the actions following this one.

   - *pattern* is the source representation of the predication that should be **joshua:ask**'ed. This is normally just the first argument to **joshua:expand-backward-rule-action**.

   - *name* is the name of a logic variable which should be bound to the query-support passed by **joshua:ask** to its continuation; this allows procedural code in the If-Part of the rule to examine the support for the various actions.

   - *truth-value* (which in the current implementation should be either **joshua:*true*** or **joshua:*false***) is the truth value which the matching predication is required to have in order to satisfy the **joshua:ask**.

   - The values of the keyword arguments to be passed to **joshua:ask**. This should normally be identical to the equivalent argument passed into **joshua:expand-backward-rule-action**.

2. `(:and` *action-descriptions*`)` This action description informs the rule compiler that the current pattern is actually a conjunction of actions all of which must be satisfied. The system-provided default method for AND predications returns this type of action description. The second element of the trigger description must be a list of action descriptions, i.e. lists returned by calling **joshua:expand-backward-rule-action**.

3. `(:or` *action-descriptions*`)` This action description informs the rule compiler that the current pattern is actually a disjunction of actions any one of which must be satisfied in order to satify the whole action. The system provided default method for OR predications returns this type of action description. The second element of the action description must be a list of action descriptions, i.e. lists returned by calling **joshua:expand-backward-rule-action**.

4. `(:procedure` *lisp-expression name*`)` This action description informs the rule compiler that the current trigger is not a pattern to be **joshua:ask**'ed but rather a Lisp expression that appears in the If-part of the backward rule. Such expressions are executed once all proceeding actions in the rule have been satisfied. The expression can act as a filter by returning either **joshua::t** or **joshua::nil**. **joshua::t** indi-

cates success; in this case the bindings accumulated up to this point are considered acceptable and rule execution continues. **joshua::nil** indicates failure; in this case the bindings are considered unacceptable.

The expression can also act as a generator in which it produces several new sets of bindings each of which is consistent with the bindings that were in effect just before the action began execution. To do this it should bind whatever logic-variables it wants to and then call **joshua:succeed**. **joshua:succeed** takes a rest-argument; the rule compiler will arrange for this value passed to **joshua:succeed** to be bound to the logic-variable which is the third element of the action description.

See the function **joshua:succeed**, page 232.

5. `(:ignore)` This action description informs the rule compiler that it should ignore this action. The are two reasons for using this type of action description. The first is to allow a rule to have actions included in it simply for the sake of clarity. The second is to include actions only to specify context.

Explain Predication Command

Traces the chain of TMS justifications for *database-predication* through rules to primitive support (premises and assumptions).

database-predication A predication object that is in the database. Must be the actual database object, and not a copy of it.

depth Specifies how many layers deep into the explanation to go before cutting off.

This is a command interface to Joshua's **joshua:explain** function.

joshua:explain *database-predication* &optional *depth (stream* ***standard-output****)* *Function*

Traces the chain of TMS justifications for *database-predication* through rules to primitive support (premises and assumptions).

database-predication A predication object that is in the database. Must be the actual database object, and not a copy of it.

depth Specifies how many layers deep into the explanation to go before cutting off.

stream Specifies a stream to which to display the output.

In general, **joshua:explain** is useful only if *database-predication* is built on some model that supports the TMS protocol.

Examples:

```
(define-predicate higher-in-food-chain (eater lower-in-food-chain)
                                        (ltms:ltms-predicate-model))
(define-predicate favorite-meal (eater food) (ltms:ltms-predicate-model))

; A good example of how to implement transitive relations

(defrule basic-food-chain (:forward)
  if [favorite-meal ?eater ?eatee]
  then [higher-in-food-chain ?eater ?eatee])

(defrule transitive-food-chain (:forward)
  if [and [favorite-meal ?eater ?eatee]
          [higher-in-food-chain ?eatee ?food]]
  then [higher-in-food-chain ?eater ?food])

(defun meals ()
  (clear)
  (tell [and [favorite-meal red-herring worm]
             [favorite-meal worm algae]])
  (tell [favorite-meal Miss-Marple red-herring] :justification :assumption)
  (cp:execute-command "Show Joshua Database"))

(meals)
True things
  [HIGHER-IN-FOOD-CHAIN MISS-MARPLE RED-HERRING]
  [HIGHER-IN-FOOD-CHAIN MISS-MARPLE WORM]
  [HIGHER-IN-FOOD-CHAIN MISS-MARPLE ALGAE]
  [HIGHER-IN-FOOD-CHAIN WORM ALGAE]
  [HIGHER-IN-FOOD-CHAIN RED-HERRING ALGAE]
  [HIGHER-IN-FOOD-CHAIN RED-HERRING WORM]
  [FAVORITE-MEAL MISS-MARPLE RED-HERRING]
  [FAVORITE-MEAL WORM ALGAE]
  [FAVORITE-MEAL RED-HERRING WORM]
False things
  None
```

```
(ask [higher-in-food-chain Miss-Marple ?food]
   #'(lambda (backward-support)
       (explain (ask-database-predication backward-support))))
[HIGHER-IN-FOOD-CHAIN MISS-MARPLE RED-HERRING] is true
  It was derived from rule BASIC-FOOD-CHAIN
  [FAVORITE-MEAL MISS-MARPLE RED-HERRING] is true
    It is an :ASSUMPTION
[HIGHER-IN-FOOD-CHAIN MISS-MARPLE WORM] is true
  It was derived from rule TRANSITIVE-FOOD-CHAIN
  [FAVORITE-MEAL MISS-MARPLE RED-HERRING] is true
    It is an :ASSUMPTION
  [HIGHER-IN-FOOD-CHAIN RED-HERRING WORM] is true
    It was derived from rule BASIC-FOOD-CHAIN
    [FAVORITE-MEAL RED-HERRING WORM] is true
      It is a :PREMISE
[HIGHER-IN-FOOD-CHAIN MISS-MARPLE ALGAE] is true
  It was derived from rule TRANSITIVE-FOOD-CHAIN
  [FAVORITE-MEAL MISS-MARPLE RED-HERRING] is true
    It is an :ASSUMPTION
  [HIGHER-IN-FOOD-CHAIN RED-HERRING ALGAE] is true
    It was derived from rule TRANSITIVE-FOOD-CHAIN
    [FAVORITE-MEAL RED-HERRING WORM] is true
      It is a :PREMISE
    [HIGHER-IN-FOOD-CHAIN WORM ALGAE] is true
      It was derived from rule BASIC-FOOD-CHAIN
      [FAVORITE-MEAL WORM ALGAE] is true
        It is a :PREMISE
```

Related Functions:

joshua:graph-tms-support

See the section "Explaining Program Beliefs" in *User's Guide to Basic Joshua*.

joshua:*false* *Variable*

A named constant used by Joshua to denote a truth value of false. You can compare truth values using **eql**.

Related Topics:

joshua:*true*
joshua:*unknown*
joshua:*contradictory*
joshua:truth-value
joshua:predication-truth-value

See the section "Truth Values" in *User's Guide to Basic Joshua*.

joshua:fetch *predication continuation* *Function*

The dual to **joshua:insert**, **joshua:fetch** is the first phase of **joshua:ask**. It takes *predication* and searches for it in the virtual database. It calls *continuation* for each occurrence of something in the database that might unify with *predication*. It is the responsibility of **joshua:ask-data** to do the unification, if that is the programmer's intent.

Note that **joshua:fetch** is required to call its continuation on objects that are actually in the database, not reconstructed copies. See **joshua:ask-data** for more discussion of this issue.

predication A pattern to search for. **joshua:fetch** must call *continuation* on a superset of the predications in the database that unify with *predication*.

continuation A function of one argument that **joshua:fetch** calls on each candidate.

For some examples: See the function **joshua:insert**, page 189.

See the section "The Joshua Rule Indexing Protocol", page 36.

Graph Forward Rule Triggers Command

Graphs the forward rule Rete network.

Forward rules or all
Graph the Rete network for which forward rules.

:Follow Extraneous Paths
Whether to include the rules which share match or merge nodes with the specified rules. This defaults to Yes.

:Orientation Draw the graph in which direction: vertical or horizontal.

:Output Destination Where to display the information.

Graph Forward Rule Triggers displays the graph of the forward rule Rete network for one or more rules. It is useful for determining the extent of node sharing between forward rules. The graph also includes a number for each node, indicating the number of environments currently held by that node. This can give you a rough measure of how much work is being done at each point in the network. See the section "Forward Rule Triggers: the Rete Network", page 27.

joshua:graph-discrimination-net *root-node* *Function*

joshua:graph-discrimination-net displays the discrimination net as a horizontal tree with the root on the leftmost side and the leaf nodes on the far right.

root-node The root node of a discrimination net, usually from the variable **ji:*data-discrimination-net***.

The different predications that discriminate to a single node are displayed individually in the leaf node.

See figure 9, for an example.

See the section "Organization of the Default Discrimination Net", page 17.

joshua:graph-query-results *backward-support* &key *(:orientation* **:vertical***) (:stream* ***standard-output****)* *Function*

A convenience function for use in an **joshua:ask** continuation. **joshua:graph-query-results** draws a graph of the support information in *backward-support*, that is, the successful query, and the reasons it succeeded.

backward-support is fully described in the dictionary entry for **joshua:ask**. **joshua:graph-query-results** both extracts and interprets the information for you.

backward-support	A support argument passed by **joshua:ask** to a continuation.
:*orientation*	Specifies the graph orientation. Default is vertical.
:*stream*	The stream on which the graph is output. Default is ***standard-output***.

The convenience function **joshua:print-query-results** prints the same information as **joshua:graph-query-results**.

The accessor function **joshua:ask-derivation** extracts all the support for a satisfied query but without interpreting it. For the sake of comparison we'll use the same examples to illustrate all three of these functions.

Examples: First, a query satisfied from the database. The graph shows the database predication that matched the query.

```
(define-predicate edible (object))
(define-predicate is-food (object))
(define-predicate contains (object substance))
(define-predicate sweet (object))
(define-predicate type-of (object type))

(tell [edible chocolate-coated-ants])
(tell [contains chocolate-coated-ants honey])
```

```
⇨ (ask [edible =what] #'graph-query-results)
          Database
[EDIBLE CHOCOLATE-COATED-ANTS]
```

The next example shows the support for a query that is satisfied from rules. We have a rule, `dessert?`, that determines if a given food is a

dessert. Each of this rule's subgoals is derived from other rules. Here are the rule definitions.

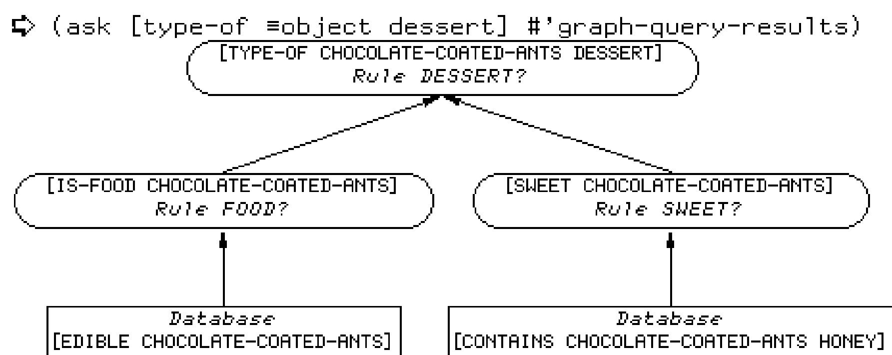
```
(defrule food? (:backward)
  if [edible  ?object]
  then [is-food  ?object])

(defrule sweet? (:backward)
  if [or [contains ?object chocolate]
         [contains ?object sugar]
         [contains ?object honey]]
  then [sweet ?object])

(defrule dessert? (:backward)
  if [and [is-food ?object]
          [sweet ?object]]
  then [type-of ?object dessert])
```

Now we **joshua:ask** what foods qualify as desserts and why. In the graph, ovals denote queries that were *not* satisfied directly by the database. Rectangles denote queries that were satisfied by the database.

The top of the graph shows the satisfied goal, and names the rule that satisfied it. The rest of the graph shows successive subgoals and how each was satisfied.

Since backward chaining stops when it finds database predications, the bottom leaves of the graph tree are queries that were satisfied by the database. Hence they are rectangles, whereas intermediate nodes are ovals.

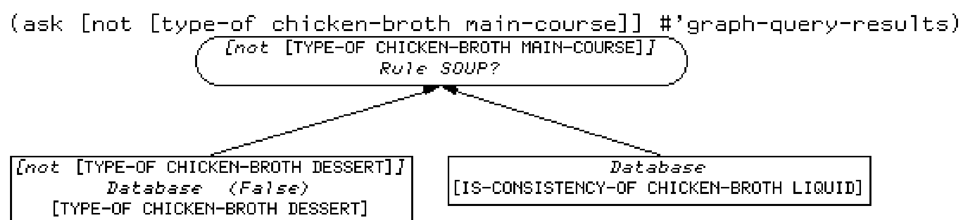
The arrows move in the *if-then* (logical conclusion) direction.



Here's an extension to the previous example, to show how the graph displays truth values of **joshua:*false***. We add a rule to eliminate first course choices: the rule says that things that are liquid and are not desserts are not a main course.

```
(define-predicate is-consistency-of (food consistency))
```

```
(defrule soup? (:backward)
  if [and [not [type-of ?food dessert]]
          [is-consistency-of ?food liquid]]
  then [not [type-of ?food main-course]])

(tell [not [type-of chicken-broth dessert]])
(tell [is-consistency-of chicken-broth liquid])
```



The graph displays the satisfied query prefixed by `[not ...]`. The database predication matching the query appears without the prefix, just as it would in the database display. The label above it indicates that its truth value is **joshua:*false***. (Predications with a truth value of **joshua:*true*** are not labelled as such in the graph Database heading.)

Related Functions:

joshua:ask
joshua:print-query-results

See the section "Querying the Database" in *User's Guide to Basic Joshua*. See the section "Explaining Backward Chaining Support" in *User's Guide to Basic Joshua*.

joshua:graph-tms-support &rest *predications* *Function*

Displays a graph of the TMS support for *predications*, that is, of the dependency information which a Truth Maintenance System stores in the database along with *predications*. The graph traces the support chain through the dependency records created by forward rules (or other callers of **joshua:justify** such as the the **:justification** keyword argument to **joshua:tell**) to the underlying primitive support (assumptions and premises). (Backward chaining support is not graphed, since the rule result is not stored in the database. For that, you probably want **joshua:graph-query-results**.)

Example:

```
(define-predicate dreams-in (language dreamer) (ltms:ltms-predicate-model))
(define-predicate counts-in (language person) (ltms:ltms-predicate-model))
(define-predicate native-speaker-of (language speaker)
                  (ltms:ltms-predicate-model))
```

```
(defrule native-speaker? (:forward)
  if [and [dreams-in ?language ?person]
          [counts-in ?language ?person]]
  then [native-speaker-of ?language ?person])

(tell [dreams-in Spanish Violet] :justification :assumption)
(tell [counts-in Spanish Violet])

Show Joshua Database (matching pattern [default All])
  [native-speaker-of ?x ?y] (opposite truth-value too? [default Yes]) Yes
True things
  [NATIVE-SPEAKER-OF SPANISH VIOLET]
False things
  None
```

You must give **joshua:graph-tms-support** the actual predication object that resides in the database, rather than a copy of it. In our example we retrieve the predication object by clicking the mouse over it in the database display.

Since the support graph traces the support for facts that are in the database, all nodes are rectangles. (Compare the display of **joshua:graph-query-results**.) The top of the graph tree shows the predication whose support we want to know about. We see that this predication was derived from a forward rule, which in turn was derived from some predications. The bottom leaves of the graph tree show primitive support (premise or assumption) denoting the end of the forward chaining process. The arrows point in the *if-then* (logical conclusion) direction.

Here's an example showing the support graph for a predication whose truth value is **joshua:*false***.

```
(define-predicate has-ticket (claimant)(ltms:ltms-predicate-model))
(define-predicate admissible (claimant)(ltms:ltms-predicate-model))
```

```
(defrule no-free-lunch (:forward)
  if [not [has-ticket ?x]]
  then [not [admissible ?x]])

(tell [not [has-ticket Jane]])
```

```
⇨ Show Joshua Database (matching pattern [default All]) All
True things
  None
False things
  [ADMISSIBLE JANE]
  [HAS-TICKET JANE]
⇨
⇨ (graph-tms-support ¬[ADMISSIBLE JANE])
      [ADMISSIBLE JANE]
(False)  Rule NO-FREE-LUNCH
             |
   [HAS-TICKET JANE]
(False)   :PREMISE

NIL
⇨
```

Predications with a truth value of **joshua:*false*** appear with an indication that they are false.

See the section "Explaining Program Beliefs" in *User's Guide to Basic Joshua*.

joshua:insert *predication* *Function*

This is the first step used by **tell**. It takes *predication* and puts it into the virtual database. It does not deal with any justification or forward rule-triggering issues. **joshua:insert** returns two values:

1. The canonical version of *predication* that is stored in the database. (That can be distinct from *predication* if another predication that is a **joshua:variant** of *predication* has previously been **joshua:insert**ed. The one already in the database is returned.) See the function **joshua:variant**, page 252.

2. A flag that indicates whether *predication* was already in the database. If the predication is was not in the database then this value should be **t**, (indicating that an insertion did, in fact, take place).

joshua:insert and **joshua:fetch** are probably methods you will want to define often in your data models, as they control the way your predications are stored. See the example developed in the section "Customizing the Data Index".

joshua:justify *conclusion truth-value* &optional *mnemonic true-support false-support unknown-support* *Function*

Sets the truth-value of things that go into the database and gives a TMS the information necessary for maintaining dependencies. For predications that implement a TMS, **joshua:justify** is the protocol function that builds and installs the justification.

conclusion	The predication being justified.
truth-value	Should be one of **joshua:*true***, **joshua:*false***, **joshua:*unknown***, or **joshua:*contradictory***. If the justification is active, then the conclusion will assume this truth-value.
mnemonic	An informative term. If the justification is being used to record the actions of a rule, then it is conventional to provide the name of the rule as the mnemonic. Justifications built by the rule-compiler follow this convention. Some TMS's may use the mnemonic to distinguish specially understood types of justifications such as premises.
true-support	A list of database predications, all of which must have truth-value **joshua:*true*** for the justification to be active.
false-support	A list of database predications, all of which must have truth-value **joshua:*false*** for the justification to be active.
unknown-support	A list of database predications, all of which must have truth-value **joshua:*unknown*** for the justification to be active. Some TMS's (e.g. the **LTMS**) may require this argument to **nil**.

If all the predications in the set of true-support have truth-value **joshua:*true***, all the predications in the false-support have truth-value **joshua:*false*** and all the predications in the unknown-support have truth-value **joshua:*unknown***, then the justification is considered to be *active*. An active justification causes the conclusion to assume its desired truth-value.

Examples:

Suppose you want to find all the `is-exiled-from` statements in your database and add a new justification to them. For example, your database might contain:

```
(define-predicate is-exiled-from (person place) (ltms:ltms-predicate-model))

(tell [is-exiled-from Prospero Padua])
(tell [is-exiled-from Henry-James US])
```

```
  Show Joshua Database (matching pattern [default All]) All
True things
  [IS-EXILED-FROM HENRY-JAMES US]
  [IS-EXILED-FROM PROSPERO PADUA]
False things
  None

  (justify [IS-EXILED-FROM HENRY-JAMES US] *true* :assumption)
NIL


  (explain [IS-EXILED-FROM HENRY-JAMES US])
[IS-EXILED-FROM HENRY-JAMES US] is true
  It is a :PREMISE
NIL

  (unjustify [IS-EXILED-FROM HENRY-JAMES US])
NIL

  (explain [IS-EXILED-FROM HENRY-JAMES US])
[IS-EXILED-FROM HENRY-JAMES US] is true
  It is an :ASSUMPTION
NIL
```

and you want to add an **:assumption** justification to each of those. You would use **joshua:justify** and **joshua:ask** as follows:

```
(ask [is-exiled-from ? ?]
     #'(lambda (backward-support)
         (justify (ask-database-predication backward-support)
                  (ask-query-truth-value backward-support)
                  :assumption))
     :do-backward-rules nil)
```

Related Functions:

joshua:unjustify

See the section "Justification and Truth Maintenance" in *User's Guide to Basic Joshua*. See the section "The Truth Maintenance Protocol", page 54.

joshua:known *proposition* *Joshua Predicate*

This modal operator checks if *proposition* is known to be either **joshua:*true*** or **joshua:*false***.

proposition A Joshua predication pattern to match.

```
The query:       (ask [known [foo ?x]] #' ...)
Succeeds when:   either [foo ?x] or [not [foo ?x]] succeed
```

If successful, **joshua:known** calls the continuation on the instantiated query.

Examples:

We use the predicate shape-of and the statements about shapes that we used to illustrate the predicate **joshua:provable**. Here they are.

```
(define-predicate shape-of (object shape))

(tell [and [shape-of door oval]
           [not [shape-of leaf pointed]]])
[AND [SHAPE-OF DOOR OVAL] [NOT [SHAPE-OF LEAF POINTED]]]

  Show Joshua Database
True things
  [SHAPE-OF DOOR OVAL]
False things
  [SHAPE-OF LEAF POINTED]]
```

The database contains one statement about shapes that is **joshua:*true*** and one that is **joshua:*false***. **joshua:known** succeeds in each case, returning the instantiated query. Note that there is no indication of truth value in the instantiated query. That is because when we ask if something is **joshua:known**, we are interested only in the existence of an answer, not in its particular truth value. (*backward-support* for the **joshua:ask** does indicate what the truth value of the instantiated query was.)

```
  (ask [known [shape-of ?object ?shape]] #'print-query)
[KNOWN [SHAPE-OF DOOR OVAL]]
[KNOWN [SHAPE-OF LEAF POINTED]] ; argument was actually false
```

A more interesting question is to ask whether a predication is *not* known to Joshua.

```
The query:        (ask [not [known [foo ?x]]] #' ...)
Succeeds when:    [foo ?x] and [not [foo ?x]] both fail
```

Examples:

```
; The proposition is not in the database or in rules
  (ask [not [known [shape-of nose pointed]]] #'print-query)
[not [KNOWN [SHAPE-OF NOSE POINTED]]]
```

joshua:known can also be used in backward rules. The goal of the very inconsiderate rule in the next example is to select a dancing partner. The rule filters out those whose ability at ?activity is unknown, keeping those who are good or bad.

```
(define-predicate need-a-partner (activity))
(define-predicate is-good-at (activity person))
(define-predicate use-as-partner (person activity))
```

```
(defrule two-left-feet-will-do (:backward)
  if [and [need-a-partner ?activity]
          [known [is-good-at ?activity ?person]]]
  then [use-as-partner ?person ?activity])

(defun test-known ()
  (clear)
  (tell [and [need-a-partner dancing]
             [is-good-at dancing Tom]
             [not [is-good-at dancing Fred]]])
  'Done.)

(test-known)
DONE.

  (ask [use-as-partner ?person ?activity] #'print-query)
[USE-AS-PARTNER TOM DANCING]
[USE-AS-PARTNER FRED DANCING]
```

The goal of the rule in the next example is to hire an applicant if his/her qualifications are excellent, even if nothing is known about the applicant's experience level.

```
(define-predicate has-qualifications (person qualifications))
(define-predicate previous-experience (person experience))
(define-predicate hire-candidate (name))

(tell [and [has-qualifications Fred poor]
           [has-qualifications Joan excellent]])
[AND [HAS-QUALIFICATIONS FRED POOR] [HAS-QUALIFICATIONS JOAN EXCELLENT]]

(defrule inexperience-no-obstacle (:backward)
  if [and [has-qualifications ?applicant excellent]
          [not [known [previous-experience ?applicant ?how-much]]]]
  then [hire-candidate ?applicant])

(ask [hire-candidate Fred] #'print-query)

(ask [hire-candidate ?applicant] #'print-query)
[HIRE-CANDIDATE JOAN]
```

Related Predicate:

joshua:provable

joshua:locate-backward-question-trigger *predication truth-value continuation context question-name* *Generic Function*

predication	A pattern under which to index a backward question.
truth-value	The truth value of the pattern under which the question should be indexed.
continuation	A function passed in which can determine whether a new question trigger is necessary.
context	Useful in advanced modeling applications.
question-name	The name of the backward-question being indexed.

Tailoring of backward-question indexing is usually accomplished by providing methods for the **joshua:locate-backward-question-trigger** and **joshua:map-over-backward-question-triggers** protocol functions. The **joshua:add-backward-question-trigger** and **joshua:delete-backward-question-trigger** methods provided as Joshua's defaults call **joshua:locate-backward-question-trigger** as a subroutine. All of the interesting tailoring of their behavior can be obtained by providing a **joshua:locate-backward-question-trigger** method.

However, it might be useful in some applications to provide **:before** or **:after** methods for the add and delete methods, for example to keep track of what questions are currently indexed where. The **joshua:locate-backward-question-trigger** method is responsible for managing the data structures used to index backward question triggers. Each backward chaining question has a unique trigger structure, indexed by the pattern (and its truth value) of the question. Just as **joshua:insert** maps variant predications to a unique location in a data index, **joshua:locate-backward-question-trigger** locates the unique place in a question index where Joshua stores a backward chaining question's trigger structure.

To accomplish this, the **joshua:locate-backward-question-trigger** method is required to follow a rather complicated pattern of behavior. This pattern is divided into four parts:

1. Using *predication* and *truth-value* it should determine where the trigger should be stored. This location should contain either **joshua::nil** or a list of backward question triggers (we'll call this the *trigger set*).

2. *continuation* should be called with the *trigger set* as an argument. It will return three values:

    a. A *new trigger set* which includes a backward question trigger data-structure corresponding to *predication* and *truth-value* (the pattern under which this trigger is indexed).

    b. A *flag* indicating whether a new trigger data-stucture was added to the *trigger set*. If this value is **joshua::t** then *trigger set* did not

already contain a backward question trigger data-structure for *predication* with truth value *truth-value*.

c. The *canonical trigger* which is the unique backward question trigger for this question.

3. If the value of *flag* is **joshua::t**, then **joshua:locate-backward-question-trigger** should update its index so that the location which used to contain *trigger set* will now contain *new trigger set*. During this step **joshua:locate-backward-question-trigger** may take whatever actions it likes to optimize the question index.

4. The method should return *canonical trigger* as its value.

The reason for this complicated pattern of behavior is as follows: **joshua:locate-backward-question-trigger** is used as a subroutine of both **joshua:add-backward-question-trigger** and **joshua:delete-backward-question-trigger**. Knowledge of how to index a pattern is localized in the **joshua:locate-backward-question-trigger** methods, while the knowledge of the internal structure of the backward trigger data-structures is localized in **joshua:add-backward-question-trigger** and **joshua:delete-backward-question-trigger**. These two higher levels routines call **joshua:locate-backward-question-trigger** passing to it *continuation*, a function which understands how to manipulate sets of backward question trigger data-structures.

Continuation adds (or deletes) a backward question trigger data-structure for the current question (if necessary) and returns enough information so that **joshua:locate-backward-question-trigger** will know what actions were taken.

joshua:locate-backward-question-trigger should return *canonical trigger* as its value so the question's debugging information can point to the actual data structure corresponding to its trigger patterns.

As an example, consider the following method which indexes backward question triggers on the property list of the predicate in the pattern.

```
(define-predicate-model predicate-backward-question-indexing () ())
```

```
(define-predicate-method
  (locate-backward-question-trigger predicate-question-indexing)
  (truth-value continuation ignore ignore)
  ;; This is part one, locate the current trigger set
  (let ((old-triggers (get (predication-predicate self)
                           'backward-question-triggers)))
    ;; part two, call the continuation
    (multiple-value-bind (new-triggers changed-p node)
        (funcall continuation old-triggers)
      ;; part three, update the index with new triggers, if something changed
      (when changed-p
        (setf (get (predication-predicate self)
                   'backward-question-triggers) new-triggers))
      ;; part four, return the canonical backward question trigger
      node)))

;;; This map method finds the triggers stored by the previous guy.
(define-predicate-method
  (map-over-backward-question-triggers predicate-question-indexing)
  (continuation)
  ;; how to collect all backward triggers that might be interested in me
  (declare (sys:downward-funarg continuation))  ;backward reference
  (loop for rete-node in (get (predication-predicate self)
                              'backward-question-triggers)
        doing (funcall continuation Rete-node)))
```

The *context* argument is provided to allow the **joshua:locate-backward-question-trigger** method to use a context sensitive indexing technique.

See the section "The Joshua Question Indexing Protocol", page 48.

joshua:locate-backward-rule-trigger *predication truth-value continuation context rule-name* — *Generic Function*

predication	A pattern under which to index a backward rule.
truth-value	The truth value of the pattern under which the rule should be indexed.
continuation	A function passed in which can determine whether a new rule trigger is necessary.
context	The entire *if*-part of the rule. Useful in advanced modeling applications. *rule-name* The name of the backward rule being indexed.

Tailoring of backward rule indexing is usually accomplished by providing methods for the **joshua:locate-backward-rule-trigger** and **joshua:map-over-backward-rule-triggers** protocol functions. The **joshua:add-backward-rule-trigger** and **joshua:delete-backward-rule-trigger** methods provided as

Joshua's defaults call **joshua:locate-backward-rule-trigger** as a subroutine. All of the interesting tailoring of their behavior can be obtained by providing a **joshua:locate-backward-rule-trigger** method.

However, it might be useful in some applications to provide **:before** or **:after** methods for the add and delete methods, for example to keep track of what rules are currently indexed where.

The **joshua:locate-backward-rule-trigger** method is responsible for managing the data structures used to index backward rule triggers. Each backward chaining rule has a unique trigger structure, indexed by the pattern (and its truth value) of the *then*-part of the rule. Just as **joshua:insert** maps variant predications to a unique location in a data index, **joshua:locate-backward-rule-trigger** locates the unique place in a rule index where Joshua stores a backward chaining rule's trigger structure.

To accomplish this, the **joshua:locate-backward-rule-trigger** method is required to follow a rather complicated pattern of behavior. This pattern is divided into four parts:

1. Using *predication* and *truth-value* it should determine where the trigger should be stored. This location should contain either **joshua::nil** or a list of backward rule triggers (we'll call this the *trigger set*).

2. *continuation* should be called with the *trigger set* as an argument. It will return three values:

    a. A *new trigger set* which includes a backward rule trigger data-structure corresponding to *predication* and *truth-value* (the pattern under which this trigger is indexed).

    b. A *flag* indicating whether a new trigger data-stucture was added to the *trigger set*. If this value is **joshua::t** then *trigger set* did not already contain a backward rule trigger data-structure for *predication* with truth value *truth-value*.

    c. The *canonical trigger* which is the unique backward rule trigger for this rule.

3. If the value of *flag* is **joshua::t**, then **joshua:locate-backward-rule-trigger** should update its index so that the location which used to contain *trigger set* will now contain *new trigger set*. During this step **joshua:locate-backward-rule-trigger** may take whatever actions it likes to optimize the rule index.

4. It should return *canonical trigger* as its value.

The reason for this complicated pattern of behavior is as follows: **joshua:locate-backward-rule-trigger** is used as a subroutine of both **joshua:add-backward-rule-trigger** and **joshua:delete-backward-rule-**

trigger. Knowledge of how to index a pattern is localized in the **joshua:locate-backward-rule-trigger** methods, while the knowledge of the internal structure of the backward trigger data-structures is localized in **joshua:add-backward-rule-trigger** and **joshua:delete-backward-rule-trigger**. These two higher levels routines call **joshua:locate-backward-rule-trigger** passing to it *continuation*, a function which understands how to manipulate sets of backward rule trigger data-structures.

Continuation adds (or deletes) a backward rule trigger data-structure for the current rule (if necessary) and returns enough information so that **joshua:locate-backward-rule-trigger** will know what actions were taken.

joshua:locate-backward-rule-trigger should return *canonical trigger* as its value so the rule's debugging information can point to the actual data structure corresponding to its trigger patterns.

As an example, consider the following method which indexes backward rule triggers on the property list of the predicate in the pattern.

```
(define-predicate-model predicate-backward-rule-indexing () ())

(define-predicate-method (locate-backward-rule-trigger predicate-rule-indexing)
                         (truth-value continuation ignore ignore)
  ;; This is part one, locate the current trigger set
  (let ((old-triggers (get (predication-predicate self) 'backward-rule-triggers)))
    ;; part two, call the continuation
    (multiple-value-bind (new-triggers changed-p node)
        (funcall continuation old-triggers)
      ;; part three, update the index with new triggers, if something changed
      (when changed-p
        (setf (get (predication-predicate self) 'backward-rule-triggers) new-triggers))
      ;; part four, return the canonical backward rule trigger
      node)))

;;; This map method finds the triggers stored by the previous guy.
(define-predicate-method (map-over-backward-rule-triggers predicate-rule-indexing)
                         (continuation)
  ;; how to collect all backward triggers that might be interested in me
  (declare (sys:downward-funarg continuation))  ;backward reference
  (loop for rete-node in (get (predication-predicate self) 'backward-rule-triggers)
       doing (funcall continuation Rete-node)))
```

The *context* argument is provided to allow the **joshua:locate-backward-rule-trigger** method to use a context sensitive indexing technique. For example, consider the following backward rule which describes the behavior of an adder:

```
(defrule adder-behavior (:backward)
 If [and
     [type-of ?a adder]
     [status-of ?a working]
     [value-of input-1 ?a ?input-1]
     [value-of input-2 ?a ?input-2]
     (unify ?sum (+ ?input-1 ?input-2))]
 Then [value-of output ?a ?sum])
```

It might be appropriate to use an *object-oriented* set of data structures to manage the indexing of this rule's trigger data structures. In such a scheme, there is one object representing the class of all adders and an additional object for each specific adder being reasoned about. The triggers for the rule should be attached to the object representing the class of all adders, since this is information shared by all the individual adders. Consider what happens when **joshua:locate-backward-rule-trigger** is called to index this rule under the pattern `[value-of output ?a ?sum]`. It should attach the corresponding trigger data-structure to the `SUM` slot of the object representing the class of all adders. However, it cannot determine this without knowing that this is a rule about adders and that information is contained in the pattern [type-of ?a adder]. It is for this reason that the entire *if*-part of the rule is passed into **joshua:locate-backward-rule-trigger**

See the section "The Joshua Rule Indexing Protocol", page 36.

joshua:locate-forward-rule-trigger *predication truth-value continuation context rule-name* *Generic Function*

predication	A pattern under which to index a forward rule trigger.
truth-value	The truth value under which the rule should be indexed.
continuation	A function passed in which can determine whether a new rule trigger is necessary.
context	The entire *if*-part of the rule. Useful in advanced modeling applications.
rule-name	The name of the rule being indexed.

Tailoring of forward rule indexing is usually accomplished by providing methods for the **joshua:locate-forward-rule-trigger** and **joshua:map-over-forward-rule-triggers** protocol functions. The **joshua:add-forward-rule-trigger** and **joshua:delete-forward-rule-trigger** methods provided as Joshua's defaults call **joshua:locate-forward-rule-trigger** as a subroutine. All of the interesting tailoring of their behavior can be obtained by providing a **joshua:locate-forward-rule-trigger** method.

However, it might be useful in some applications to provide **:before** or **:after** methods for the add and delete methods, for example to keep track of what rules are currently indexed where.

The **joshua:locate-forward-rule-trigger** method is responsible for managing the data structures used to index forward rule triggers. (In Joshua forward rule triggers serve the role of match nodes in a *Rete Network*). In general, Joshua tries to share forward rule triggers as much as possible. If the same pattern appears in the *IF* part of two forward chaining rules, Joshua tries to use the same forward rule trigger for both occurances of the pattern. (By the pattern we mean two predications which are **joshua:variant**s of each other and which have the same truth value.) Thus just as **joshua:insert** is should map variant predications to the same location in a data model, **joshua:locate-forward-rule-trigger** should map **joshua:variant** patterns to the same location in a rule index.

To accomplish this, the **joshua:locate-forward-rule-trigger** method is required to follow a rather complicated pattern of behavior. This pattern is divided into four parts:

1. Using *predication* and *truth-value* it should determine where the trigger should be stored. This location should contain either **joshua::nil** or a list of forward rule triggers (we'll call this the *trigger set*).

2. *continuation* should be called with the *trigger set* as an argument. It will return 3 values:

   a. A *new trigger set* which includes a forward rule trigger data-structure corresponding to *predication* and *truth-value* (the pattern under which this trigger is indexed).

   b. A *flag* indicating whether a new trigger data-stucture was added to the *trigger set*. If this value is **joshua::t** then *trigger set* did not already contain a forward rule trigger data-structure for *predication* with truth value *truth-value*.

   c. The *canonical trigger* which is the unique forward rule trigger data-structure which for the pattern *predication* with truth value *truth-value*.

3. If the value of *flag* is **joshua::t**, then **joshua:locate-forward-rule-trigger** should update its index so that the location which used to contain *trigger set* will now contain *new trigger set*. During this step **joshua:locate-forward-rule-trigger** may take whatever actions it likes to optimize the rule index.

4. The method should return *canonical trigger* as its value.

The reason for this complicated pattern of behavior is as follows: **joshua:locate-forward-rule-trigger** is used as a subroutine of both **joshua:add-forward-rule-trigger** and **joshua:delete-forward-rule-trigger**. Knowledge of how to index a pattern is localized in the **joshua:locate-forward-rule-trigger** methods, while the knowledge of the internal struc-

ture of the forward trigger data-structures is localized in **joshua:add-forward-rule-trigger** and **joshua:delete-forward-rule-trigger**. These two higher levels routines call **joshua:locate-forward-rule-trigger** passing to it *continuation*, a function which understands how to manipulate sets of forward rule trigger data-structures.

Continuation adds (or deletes) a forward rule trigger data-structure for the current rule (if necessary) and returns enough information so that **joshua:locate-forward-rule-trigger** will know what actions were taken.

joshua:locate-forward-rule-trigger should return *canonical trigger* as its value so the rule's debugging information can point to the actual data structure corresponding to its trigger patterns.

As an example, consider the following method which indexes forward rule triggers on the property list of the predicate in the pattern.

```
(define-predicate-model predicate-forward-rule-indexing () ())

(define-predicate-method (locate-forward-rule-trigger predicate-rule-indexing)
                         (truth-value continuation ignore ignore)
  ;; This is part one, locate the current trigger set
  (let ((old-triggers (get (predication-predicate self) 'forward-rule-triggers)))
    ;; part two, call the continuation
    (multiple-value-bind (new-triggers changed-p node)
        (funcall continuation old-triggers)
      ;; part three, update the index with new triggers, if something changed
      (when changed-p
        (setf (get (predication-predicate self) 'forward-rule-triggers) new-triggers))
      ;; part four, return the canonical forward rule trigger
      node)))

;;; This map method finds the triggers stored by the previous guy.
(define-predicate-method (map-over-forward-rule-triggers predicate-rule-indexing)
                         (continuation)
  ;; how to collect all forward triggers that might be interested in me
  (declare (sys:downward-funarg continuation))  ;forward reference
  (loop for rete-node in (get (predication-predicate self) 'forward-rule-triggers)
        doing (funcall continuation Rete-node)))
```

The *context* argument is provided to allow the **joshua:locate-forward-rule-trigger** method to use a context sensitive indexing technique. For example, consider the following forward rule which describes the behavior of an adder:

```
(defrule adder-behavior (:forward)
 If [and
     [type-of ?a adder]
     [status-of ?a working]
     [value-of input-1 ?a ?input-1]
     [value-of input-2 ?a ?input-2]]
 Then `[value-of output ?a ,(+ ?input-1 ?input-2)])
```

It might be appropriate to use an *object-oriented* set of data structures to manage the indexing of this rule's trigger data structures. In such a scheme, there is one object representing the class of all adders and an additional object for each specific adder being reasoned about. The triggers for the rule should be attached to the object representing the class of all adders, since this is information shared by all the individual adders. Notice, however, that **joshua:locate-forward-rule-trigger** is called once for each trigger pattern. Consider what happens when it is called with `[value-of input-1 ?a ?input-1]` as its argument. It should attach the corresponding trigger data-structure to the `input-1` slot of the object representing the class of all adders. However, it cannot determine this without knowing that this is a rule about adders and that information is contained in the pattern [type-of ?a adder]. It is for this reason that the entire *if*-part of the rule is passed into **joshua:locate-forward-rule-trigger**.

There is a strong similarity between the role played by **joshua:locate-forward-rule-trigger** and that played by the combination of **joshua:insert** and **joshua:uninsert**. **joshua:locate-forward-rule-trigger** manages the indexing (and unindexing) of forward rules. **joshua:insert** manages the indexing of facts and **joshua:uninsert** manages the unindexing of facts. There is also a lack of symmetry in that there are two distinct methods for facts and only one method for rules. The decision to modularlize the two processes differently was based on efficiency consideration. Facts are added and removed much more frequently than rules; thus, it was felt that a slight loss of modularity would be tolerable to achieve higher performance while inserting (and removing) facts.

See the section "The Joshua Rule Indexing Protocol", page 36. See the section "Customizing the Rule Index", page 88.

joshua:logic-variable-name *logic-variable* *Function*

Returns the symbol which is the name of logic-variable.

logic-variable

An unbound logic variable.

For example:

```
(typecase x
 (unbound-logic-variable (logic-variable-name x))
 (otherwise x))
```

joshua:logic-variable-maker-p *form* *Function*

form An s-expression.

A predicate of one argument. It returns **joshua::t** if the argument is a logic-variable-maker and **joshua::nil** otherwise.

```
(setq x (read))?A
```

yields:

```
(ji::logic-variable-maker |?A|)
```

and

```
(logic-variable-maker-p x)
```

yields:

```
T
```

joshua:logic-variable-maker-name *lv-maker* *Function*

lv-maker A logic-variable-maker s-expression.

This returns the name of the logic-variable-maker. For example:

```
(read)?L
```

yields:

```
(JI::LOGIC-VARIABLE-MAKER |?L|)
```

and

```
(logic-variable-maker-name (JI::LOGIC-VARIABLE-MAKER |?L|))
```

yields

```
|?L|
```

ltms:ltms-mixin *Flavor*

This flavor provides the Joshua LTMS methods. Since it defines only TMS protocol methods, it must be combined with some model which defines the other protocol methods.

Related topics:

ltms:ltms-predicate-model
joshua:basic-tms-mixin
joshua:define-predicate-model
joshua:define-predicate

ltms:ltms-predicate-model *Flavor*

This flavor combines the Joshua LTMS behavior with the default predicate behavior. It is composed of **ltms:ltms-mixin** and **joshua:default-predicate-model**.

Related functions:

joshua:define-predicate-model
joshua:define-predicate

joshua:make-predication *statement* &optional *area* *Function*

Construct a predication out of the specified *statement* (in the optional *area* supplied). The newly constructed predication is *not* entered in the database, unless you combine **joshua:make-predication** with **joshua:tell**.

You should seldom need to know about this, as the [ ] syntax is used in Joshua contexts as a reader macro for **joshua:make-predication**.

statement	A list whose first element is the name of a (defined) predicate. The rest of the list elements are the arguments to the predicate.
area	Storage area to cons in

Examples:

```
(define-predicate shape-of (object shape))

(make-predication '(shape-of window round))
[SHAPE-OF WINDOW ROUND]    ; this is not in the database

(tell (make-predication '(shape-of window round)))
[SHAPE-OF WINDOW ROUND]  ; new predication added to the database
T
```

joshua:make-predication is useful for constructing Joshua predications from data generated within Lisp code. (Still, backquoting [ ] expressions should suffice most of the time.)

Related Functions:

joshua:define-predicate

See the section "Predications and Predicates" in *User's Guide to Basic Joshua*.

joshua:make-object *object-type* &key *:name* *Function*

This function instantiates Joshua objects.

joshua:map-over-database-predications *predication-pattern function* *Macro*

A convenience macro for **joshua:ask**. Use it whenever you want to find an answer to a query in the database without using rules or questions.

joshua:map-over-database-predications finds all database predications that unify with *predication-pattern* and applies *function* to each.

predication-pattern A pattern to match against database predications.

function Specifies the operation to do on each database predication that unifies with predication-pattern. Should be a function of one argument.

`(map-over-database-predications <predication> <continuation>)` is equivalent to:

```
(ask [foo ?x]
     #'(lambda (support)
         (funcall <cont>
                  (ask-database-predication support)))
     :do-backward-rules nil)
```

Example:

We'll build an author-title index for a library, using **joshua:tell** statements. We'll include an LTMS in our predicate definitions, so that we can later get **joshua:explain** to tell us about some database predications.

```
(define-predicate author-of (work author) (ltms:ltms-predicate-model))

(defun build-author-title-index1 ()
  (clear)
  (tell [and [author-of "The Interpretation of Dreams" Freud]
             [author-of "Hedda Gabler" Ibsen]
             [author-of "Totem and Taboo" Freud]
             [author-of "A Doll's House" Ibsen]])
  (cp:execute-command "Show Joshua Database"))
BUILD-AUTHOR-TITLE-INDEX1

(build-author-title-index1)
True things
  [AUTHOR-OF "A Doll's House" IBSEN]
  [AUTHOR-OF "Totem and Taboo" FREUD]
  [AUTHOR-OF "Hedda Gabler" IBSEN]
  [AUTHOR-OF "The Interpretation of Dreams" FREUD]
False things
  None
```

The first example looks in the library database and removes from it all of Freud's books (perhaps for rebinding due to overuse). We use **joshua:map-over-database-predications** to get our hands on the actual predication objects so that we can remove them.

To allow easy replacement of this information we'll **joshua:unjustify** the facts rather than actually removing them with **joshua:untell**. The truth value of each of these facts becomes **joshua:*unknown***, even though they

physically remain in the system.

```
(defun away-with-sigmund ()
  (map-over-database-predications [author-of ?work Freud] #'unjustify)
  (cp:execute-command "Show Joshua Database"))
AWAY-WITH-SIGMUND

(away-with-SIGMUND)
True things
  [AUTHOR-OF "A Doll's House" IBSEN]
  [AUTHOR-OF "Hedda Gabler" IBSEN]
False things
  None
```

Let's add a forward rule that says the library owns any work that was authored by Shakespeare, and then build another database.

```
(define-predicate owns-library (work) (ltms:ltms-predicate-model))

(defrule Shakespeare-holdings (:forward)
  if [author-of ?work Shakespeare]
  then [owns-library ?work])

(defun build-author-title-index2 ()
  (clear)
  (tell [and [author-of "King Lear" Shakespeare]
             [author-of "Hedda Gabler" Ibsen]
             [owns-library "Trumpeting Joshua"]
             [author-of "A Doll's House" Ibsen]])
  (cp:execute-command "Show Joshua Database"))
BUILD-AUTHOR-TITLE-INDEX2

(build-author-title-index2)
True things
  [OWNS-LIBRARY "Trumpeting Joshua"]  [AUTHOR-OF "Hedda Gabler" IBSEN]
  [OWNS-LIBRARY "King Lear"]          [AUTHOR-OF "King Lear" SHAKESPEARE]
  [AUTHOR-OF "A Doll's House" IBSEN]
False things
  None
```

We can now ask Joshua to **joshua:explain** the database predications about works the library owns.

```
(map-over-database-predications [owns-library ?work] #'explain)
[OWNS-LIBRARY "Trumpeting Joshua"] is true
  It is a :PREMISE
[OWNS-LIBRARY "King Lear"] is true
  It was derived from rule SHAKESPEARE-HOLDINGS
  [AUTHOR-OF "King Lear" SHAKESPEARE] is true
    It is a :PREMISE
```

Here's an example showing the display when the database predication has a truth value of **joshua:*false***. The predication displays without indicating its truth value; that information is supplied by the accompanying explanation.

```
(tell [not [owns-library "Everyday Sanskrit"]])
¬[OWNS-LIBRARY "Everyday Sanskrit"]
T

(map-over-database-predications [not [owns-library ?work]] #'explain)
[OWNS-LIBRARY "Everyday Sanskrit"] is false
  It is a :PREMISE
```

The accessor function **joshua:ask-database-predication** can also be used to extract database predications from the backward support supplied to the **joshua:ask** continuation. Most of the time **joshua:map-over-database-predications** probably serves just as well, and it is easier to use. For comparison we are using the same examples to illustrate both functions.

Related Functions:

joshua:ask

See the section "Querying the Database" in *User's Guide to Basic Joshua*.

joshua:map-over-backward-question-triggers *predication continuation* — *Generic Function*

predication	Is the query being **joshua:ask**ed.
continuation	Is a function of one argument. The argument passed to this function should be a backward-question-trigger.

joshua:map-over-backward-question-triggers is the Joshua protocol function is responsible for finding backward-questions capable of satisfying a query given to **joshua:ask**. It searches the question index to find a set of backward-question triggers whose patterns might unify with *predication* (*predication* is the query given to **joshua:ask**). **joshua:map-over-backward-question-triggers** calls *continuation* once for each backward-question-trigger found, thereby invoking the question.

joshua:map-over-backward-question-triggers is implemented by protocol methods (either the system supplied default or a user defined method implementing a special question-indexing model). To make such methods easier to write, all the knowledge of how to actually invoke a backward question is packaged in the *continuation* function which is passed into **joshua:map-over-backward-question-triggers** by the **joshua:ask-questions** protocol function.

joshua:map-over-backward-question-triggers is the dual protocol function to **joshua:locate-backward-question-trigger**. Both of these functions are used to manipulate the question-index. **joshua:locate-backward-question-trigger** is responsible for inserting and deleting backward-question-triggers

while **joshua:map-over-backward-question-triggers** is responsible for looking up question-triggers in response to a query.

Related Function:

joshua:locate-backward-question-trigger

See the section "The Joshua Question Indexing Protocol", page 48.

joshua:map-over-backward-rule-triggers *predication continuation* *Generic Function*

predication Is the query being **joshua:ask**ed.

continuation Is a function of one argument. The argument passed to this function should be a backward-chaining rule-trigger.

joshua:map-over-backward-rule-triggers is the Joshua protocol function which is responsible for finding backward-chaining rules capable of satisfying a query given to **joshua:ask**. It searches the rule index to find a set of backward-chaining rule triggers whose patterns might unify with *predication* (*predication* is the query given to **joshua:ask**). **joshua:map-over-backward-rule-triggers** calls *continuation* once for each backward-chaining rule-trigger found, thereby invoking the rule.

joshua:map-over-backward-rule-triggers is implemented by protocol methods (either the system supplied default or a user defined method implementing a special rule-indexing model). To make such methods easier to write, all the knowledge of how to actually invoke a backward chaining rule is packaged in the *continuation* function which is passed into **joshua:map-over-backward-rule-triggers** by the **joshua:ask-rules** protocol function.

joshua:map-over-backward-rule-triggers is the dual protocol function to **joshua:locate-backward-rule-trigger**. Both of these functions are used to manipulate the rule-index. **joshua:locate-backward-rule-trigger** is responsible for inserting and deleting backward-chaining rule-triggers while **joshua:map-over-backward-rule-triggers** is responsible for looking up rule-triggers in response to a query. See the generic function **joshua:locate-backward-rule-trigger**, page 196. See the section "The Joshua Rule Indexing Protocol", page 36. See the section "Customizing the Rule Index", page 88.

joshua:map-over-forward-rule-triggers *predication continuation* *Generic Function*

predication Is the fact being inserted into the database by **joshua:tell**.

continuation Is a function of one argument. The argument given to this function must be a forward rule trigger.

joshua:map-over-forward-rule-triggers is the Joshua protocol function which is responsible for finding forward-chaining rules which should be

triggered in response to the new *predication* being added to the virtual database by **joshua:tell**. It searches the rule index to find a set of forward-chaining rule triggers whose patterns might unify with *predication* (*predication* is the fact being inserted into the database by **joshua:tell**). **joshua:map-over-forward-rule-triggers** calls *continuation* once for each forward-chaining rule-trigger found, thereby invoking the rule.

joshua:map-over-forward-rule-triggers is implemented by protocol methods (either the system supplied default or a user defined method implementing a special rule-indexing model). To make such methods easier to write, all the knowledge of how to actually invoke a forward chaining rule is packaged in the *continuation* function which is passed into **joshua:map-over-forward-rule-triggers** by the default **joshua:tell** method.

joshua:map-over-forward-rule-triggers is the dual protocol function to **joshua:locate-forward-rule-trigger**. Both of these functions are used to manipulate the rule-index. **joshua:locate-forward-rule-trigger** is responsible for inserting and deleting forward-chaining rule-triggers while **joshua:map-over-forward-rule-triggers** is responsible for looking up rule-triggers in response to a query. See the generic function **joshua:locate-forward-rule-trigger**, page 199. See the section "The Joshua Rule Indexing Protocol", page 36. See the section "Customizing the Rule Index", page 88.

joshua:map-over-object-hierarchy *function-to-apply &optional initial-object* *Function*

Maps a function over an object and all its parts, recursively descending the part hierarchy. If the optional argument *initial-object* is not supplied then the function will be applied to all objects. If *initial-object* is supplied, then only the piece of the part hierarchy starting from that object will be mapped over.

joshua:map-over-slots-in-object-hierarchy *function-to-apply &optional initial-object* *Function*

This function is a utility provided as part of the Joshua object facility. It combines the operations provided by **joshua:map-over-object-hierarchy** and **joshua:map-over-slots-of-object**. It applies a function to all the slots of an object and its parts. When the optional argument *initial-object* is not supplied then the function will be applied to all objects.

Note that the function is applied to the slot itself, and not to the value of the slot. If the value of the slot is desired, use **joshua:slot-current-value** to get it.

joshua:map-over-slots-of-object *function-to-apply object* *Function*

This function is a utility provided as part of the Joshua object facility. It maps a function over all the slots of an object.

Note that the function is applied to the slot itself, and not to the value of the slot. If the value of the slot is desired, use **joshua:slot-current-value** to get it.

ji:model-cant-handle-query *Flavor*

This flavor is the base flavor for conditions that are signalled by **joshua:ask-data** and **joshua:fetch** to indicate that they have been passed a query which is more general than they can handle.

The Joshua Database Protocol allows you to structure your data in ways that are appropriate for your application; sometimes this involves trading off generality for performance. For example, if a significant portion of your data consists of object-attribute-value triples (such as the *color* of the *block* is *blue*), then you might want to use an object-oriented representation (such as **joshua::flavor** instances) to store this data. However, using this representation makes it awkward or slow to respond to a query that asks for every object with a specific property, such as:

```
[has-eye-color ?who blue]
```

An implementation of **joshua:ask-data** or **joshua:fetch** would ideally answer such a query even if it did so slowly. However, such queries may be of such little value to an application that a developer decides not to waste effort on implementing a method that can respond to the query.

It is important, however, that **joshua:fetch** and **joshua:ask-data** methods do not cause errors when faced with a query that they do not wish to handle. One reason for this is that the command Show Joshua Database may post such a query even if the application never makes such queries on its own.

The contract of **joshua:ask-data** and **joshua:fetch** requires these methods to **joshua::signal** a specific condition when they decline to handle a query. The base flavor for such condition objects is **ji:model-cant-handle-query**. A second condition flavor (built on this base flavor) is called **ji:model-can-only-handle-positive-queries** which (as the name suggests) should be used if the implementation is presented with a negated query, but only expects queries which are not negated.

The following is an example of how to use these conditions:

```
(define-predicate-method (ask-data object-model)
                         (truth-value continuation)
  (unless  (eql truth-value *true*)
    (signal 'ji:model-can-only-handle-positive-queries
            :query self
            :model 'port-direction-model))
  (with-statement-destructured (object value) ()
    (typecase object
      (unbound-logic-variable
        (signal 'ji:model-cant-handle-query
                :model 'port-direction-model
                :query self))
      (otherwise  < whatever you really want to do > ))))
```

ji:model-only-handles-positive-queries *Flavor*

This flavor is the flavor of condition objects that are signalled by **joshua:ask-data** and **joshua:fetch** to indicate that they have been passed a negated query when they only handle non-negated queries.

The Joshua Database Protocol allows you to structure your data in ways that are appropriate for your application; sometimes this involves trading off generality for performance. For example, if a significant portion of your data consists of object-attribute-value triples (such as the *color* of the *block* is *blue*), then you might want to use an object-oriented representation (such as **joshua::flavor** instances) to store this data. However, using this representation makes it awkward or slow to respond to a query that asks for every object with a specific property, such as:

```
[has-eye-color ?who blue]
```

An implementation of **joshua:ask-data** or **joshua:fetch** would ideally answer such a query even if it did so slowly. However, such queries may be of such little value to an application that a developer decides not to waste effort on implementing a method that can respond to the query.

It is important, however, that **joshua:fetch** and **joshua:ask-data** methods do not cause errors when faced with a query that they do not wish to handle. One reason for this is that the command Show Joshua Database may post such a query even if the application never makes such queries on its own.

The contract of **joshua:ask-data** and **joshua:fetch** requires these methods to **joshua::signal** a specific condition when they decline to handle a query. The base flavor for such condition objects is **ji:model-cant-handle-query**. A second condition flavor (built on this base flavor) is called **ji:model-can-only-handle-positive-queries** which (as the name suggests) should be used if the implementation is presented with a negated query, but only expects queries which are not negated.

The following is an example of how to use these conditions:

```
(define-predicate-method (ask-data object-model)
                         (truth-value continuation)
  (unless  (eql truth-value *true*)
    (signal 'ji:model-can-only-handle-positive-queries
            :query self
            :model 'port-direction-model))
  (with-statement-destructured (object value) ()
    (typecase object
      (unbound-logic-variable
        (signal 'ji:model-cant-handle-query
                :model 'port-direction-model
                :query self))
      (otherwise  < whatever you really want to do > ))))
```

joshua:negate-truth-value *truth-value* &optional (*if-unknown* **joshua:*unknown***) *Function*

Negates a numeric *truth-value*. That is, **joshua:negate-truth-value** turns **joshua:*true*** into **joshua:*false*** and vice-versa.

truth-value
: An integer truth value, which must be one of **joshua:*true***, **joshua:*false***, or **joshua:*unknown***.

if-unknown
: The value to return if *truth-value* is **joshua:*unknown***.

Related Presentation Type:

joshua:truth-value

See the section "Truth Values" in *User's Guide to Basic Joshua*.

joshua:no-variables-in-data-mixin *Flavor*

This is a predicate model which may be mixed into the definition of any predicate.

For example,

```
(define-predicate sick-with (person disease)
   (no-variables-in-data-mixin default-predicate-model))
```

If one attempts to **joshua:tell** such a predication and if the predication contains unbound logic-variables, the an error is signalled. For example:

```
(tell [sick-with ?x cholera])

Error: Trying to TELL [SICK-WITH ?X CHOLERA]
       which contains logic-variables
```

Therefore, the system can safely assume that any database predication of type **joshua:no-variables-in-data-mixin** contains only explicit data.

All the predicates used by "The Joshua Object Facility" include **joshua:no-variables-in-data-mixin** so most rules that refer only to data within the object facility will be optimized automatically. These predicates are:

- **joshua:part-of-mixin**
- **joshua:part-of-mixin**
- **joshua:value-of**
- **ltms:value-of**
- **joshua:object-type-of**
- **ltms:object-type-of**

- **joshua:equated**
- **ltms:equated**

joshua:nontrivial-tms-p *predication* *Generic Function*

Returns either **t** or **nil** to indicate whether *predication* is based on a flavor (e.g. **ltms:ltms-mixin**) that supports a TMS. A return value of **t** means that *predication* does contain TMS information.

See the section "The Truth Maintenance Protocol", page 54.

joshua:notice-truth-value-change *database-predication old-truth-value* *Function*

Called whenever the truth-value of *predication* changes from *old-truth-value* to some new truth-value.

database-predication A predication

old-truth-value The truth value that just changed

The new truth-value is available in the predication by the time notice-truth-value-change is called. It can be examined using **joshua:predication-truth-value**.

This protocol function allows you to update data structures that depend on the truth value of a predication as the truth values change. (You might want to do that, for example, in advanced uses of modeling.)

See the sections on "Signalling Truth Value Changes" and **joshua:act-on-truth-value-change**

joshua:object-type-of *object type* *Joshua Predicate*

This predicate is part of the Joshua object facility. It is used to query the Joshua object type hierarchy. It is nearly always the predicate of the first predication in the triggers of a rule that refers to objects.

joshua:object-type-of is an ask-only predicate. A predication with **joshua:object-type-of** as its predicate cannot be an argument to **joshua:tell**.

joshua:object-type-of is built using **joshua:type-of-mixin**.

ltms:object-type-of *object type* *Joshua Predicate*

This predicate is part of the Joshua object facility. It is used in the same manner as **joshua:object-type-of**. Because rules whose triggers are all TMS predications may appear cleaner or more uniform than rules which mix TMS and non-TMS predications, **ltms:object-type-of** is supplied so that rules employing other TMS predications may refer to type relationships and keep their uniform appearance.

joshua:part-of *superpart-object subpart-object* *Joshua Predicate*

This predicate is part of the Joshua object facility. It is used to query the Joshua part hierarchy about part relationships. **joshua:part-of** is an ask-only predicate; it cannot be used in **joshua:tell**.

joshua:part-of is built using **joshua:part-of-mixin**.

ltms:part-of *superpart-object subpart-object* *Joshua Predicate*

This predicate is part of the Joshua object facility. It is used in the same manner as **joshua:part-of**. Because rules whose triggers are all TMS predications may appear cleaner or more uniform than rules which mix TMS and non-TMS predications, **ltms:part-of** is supplied so that rules employing other TMS predications may refer to part relationships and keep their uniform appearance.

ltms:part-of is built using **joshua:part-of-mixin**.

joshua:part-of-mixin *Flavor*

This flavor-mixin is part of the Joshua object facility. It may be used to add part-whole behaviour, like that of the default part-whole predicate **joshua:part-of**, to predicate models defined by the user.

joshua:part-of-mixin inherits from **joshua:tell-error-model** and **joshua:ask-data-only-mixin**.

joshua:positions-forward-rule-matcher-can-skip *rule-trigger* *Generic Function*

rule-trigger	The source representation of a forward rule trigger. See the section "The Source Representaton of Predications and Logic-variables".

The protocol function **joshua:positions-forward-rule-matcher-can-skip** is used to improve the efficiency of the match function generated by the forward rule compiler. It informs the rule compiler that it need not emit checking code for certain positions in the pattern *predication*, allowing the rule compiler to generate a shorter and more efficient matcher. The positions that can be skipped are exactly those which can be guaranteed to have been checked by the rule indexer. **joshua:positions-forward-rule-matcher-can-skip** returns a list of the positions that can be skipped by the match compiler.

For example, suppose that we are using a forward-rule indexing scheme in which the trigger for each pattern of the rule is stored on the property-list of the predicate symbol of the pattern.

```
(define-predicate-model predicate-forward-rule-indexing () ())
```

```
(define-predicate-method (locate-forward-rule-trigger predicate-rule-indexing)
                         (truth-value continuation ignore ignore)
  ;; This is part one, locate the current trigger set
  (let ((old-triggers (get (predication-predicate self) 'forward-rule-triggers)))
    ;; part two, call the continuation
    (multiple-value-bind (new-triggers changed-p node)
        (funcall continuation old-triggers)
      ;; part three, update the index with new triggers, if something changed
      (when changed-p
        (setf (get (predication-predicate self) 'forward-rule-triggers) new-triggers))
      ;; part four, return the canonical forward rule trigger
      node)))

;;; This map method finds the triggers stored by the previous guy.
(define-predicate-method (map-over-forward-rule-triggers predicate-rule-indexing)
                         (continuation)
  ;; how to collect all forward triggers that might be interested in me
  (declare (sys:downward-funarg continuation))  ;forward reference
  (loop for rete-node in (get (predication-predicate self) 'forward-rule-triggers)
        doing (funcall continuation Rete-node)))
```

When we **joshua:tell** a predication whose **predicate** is that same as that in *predication*, the **joshua:map-over-forward-rule-triggers** method will only retrieve triggers for patterns which have this same **predicate**. The continuation called by **joshua:map-over-forward-rule-triggers** will then call the matcher generated by the forward-rule compiler. Clearly this matching function need not check that the first symbol in *predication* matches the first symbol in predication just **joshua:insert**ed by **joshua:tell**, since the **joshua:map-over-forward-rule-triggers** has just done so.

The return value of **joshua:positions-forward-rule-matcher-can-skip** is a list of *positions* that can be skipped by the match compiler. The list of *positions* consists of sublists of *predication*; the **joshua::car** of each of these sublists is a token for which the matcher need generate no code. For example, if we use an indexing scheme which guarantees that every symbol in a pattern is checked by the indexer, then the **joshua:positions-forward-rule-matcher-can-skip** method should return a list of every sublist of the pattern which begins with a symbol:

```
[Foobar a ?x b c ?y]  →

((foobar a ?x b c ?y)
 (a ?x b c ?y)
 (b c ?y)
 (c ?y))
```

The default method for the **joshua:positions-forward-rule-matcher-can-skip** protocol function skips every symbol in the pattern, since the default

indexer uses every symbol in the pattern. If you create an indexing scheme of your own which does not check every symbol then you must provide a method for this protocol function or your forward rules may get incorrectly compiled. Here is the method that should be provided with the example above:

```
(define-predicate-method
  (predicate-forward-rule-indexing positions-forward-rule-matcher-can-skip) ()
  (list (predication-statement self)))
```

joshua:predication *Flavor*

The non-instantiable base flavor for all predications in Joshua. It is mixed into new predications via **joshua:define-predicate**.

You can test for this flavor by using **typep** or **joshua:predicationp** (into which **typep** is optimized).

Related Presentation Types:

joshua:predication
joshua:database-predication

joshua:predication *Presentation Type*

The type for accepting or presenting a Joshua predication. When used to accept a predication from the user, this presentation type will parse the input and create a new instance of the predication. If the predication is entered by using the mouse, the parser will return the predication that the user selected. That is, it will not create a new copy of the predication. This presentation type is convenient for reading in predications, as it confirms that the predicate is defined and the arguments are correct, and reprompts until the input is a valid predication.

Example:

```
(accept 'predication)
Enter a predication: [jericho:good-to-eat bananas]
[JERICHO:GOOD-TO-EAT BANANAS]
PREDICATION
```

joshua:predication-maker-p *form* *Function*

form An s-expression.

A predicate of one argument. It returns **joshua::t** if the argument is a predication-maker and **joshua::nil** otherwise.

For example:

```
(setq x (read))[Foobar ?x a]
```

yields:

```
(ji::predication-maker
  '(foobar (ji::logic-variable-maker |?x|) a))
```

and

```
(predication-maker-p
 (ji::predication-maker
  '(foobar (ji::logic-variable-maker |?x|) a)))
```

yields:

```
T
```

joshua:predication-maker-predicate *form* *Function*

form A predication-maker s-expression.

This returns the predicate of a predication-maker form.

For example,

```
(read)[Foobar a b]
```

yields:

```
(JI::PREDICATION-MAKER '(FOOBAR A B))
```

and

```
(predication-maker-predicate
  (JI::PREDICATION-MAKER '(FOOBAR A B)))
```

yields:

```
FOOBAR
```

joshua:predication-maker-statement *form* *Function*

form A Predication-Maker list.

This returns the "statement" part of the predication-maker list structure. For example,

```
(read)[foobar a b]
```

yields:

```
(JI::PREDICATION-MAKER '(FOOBAR A B))
```

and

```
(predication-maker-statement
  (JI::PREDICATION-MAKER '(FOOBAR A B)))
```

yields:

```
(FOOBAR A B)
```

Similarly,

```
(read) `[foobar ,a b]
-> (JI::PREDICATION-MAKER `(FOOBAR ,A B))
(predication-maker-statement *)
-> (FOOBAR (#:|,| . A) B)
```

joshua:predicationp *object* *Function*

Checks whether *object* is a Joshua predication, that is, whether the object is built on the base flavor **joshua:predication**. **joshua:predication** is the root of the Joshua model tree.

joshua:predicationp returns **t** if the object is a Joshua predication, otherwise **nil**.

object An object in the Lisp world.

Examples:

```
  (define-predicate valid-word (word language))

  (tell [valid-word incarnadine English])
[VALID-WORD INCARNADINE ENGLISH]
T

  (predicationp [VALID-WORD INCARNADINE ENGLISH])
     ; click on object returned by tell
(PREDICATION FLAVOR:VANILLA)

 (ask [valid-word incarnadine ?language]
      #'(lambda (backward-support)
          (when (predicationp (ask-database-predication backward-support))
            (print (ask-database-predication backward-support)))))
[VALID-WORD INCARNADINE ENGLISH]
```

You can use **typep** to do the same test as **joshua:predicationp**. In fact, the compiler optimizes the form:

```
(typep x 'predication)
```

into the form:

```
(predicationp x)
```

For example:

```
  (ask [valid-word incarnadine ?language]
       #'(lambda (backward-support)
           (when (typep (ask-database-predication backward-support)
                        'predication)
             (print (ask-database-predication backward-support)))))
[VALID-WORD INCARNADINE ENGLISH]
```

Related Functions:

joshua:predication
typep

joshua:predication-predicate *predication* *Function*

Returns the predicate symbol of *predication*.

predication Any predication.

Related Function:

joshua:predication-statement

joshua:predication-statement *predication* *Function*

Returns the list corresponding to the statement of *predication*. The first element of the list is the predicate symbol. The rest of the list contains the arguments.

predication Any predication.

For example:

```
(define-predicate employee (name social-security-number department))

(predication-statement [employee "John Doe" 345267791 shipping])
(EMPLOYEE "John Doe" 345267791 SHIPPING)

(predication-statement [not [employee "Eve" 2 gardening]])
(NOT [EMPLOYEE "Eve" 2 GARDENING])
```

Related Functions:

joshua:make-predication
joshua:predication-predicate

joshua:predication-truth-value *predication* *Function*

Returns the numeric truth value of *predication*.

predication A Joshua predication.

Since truth value is a property of the database, the truth value of a predication not in the database is not defined. In general it will be **joshua:*unknown***.

Checking the truth value of a predication is done using **joshua:ask**. The **joshua:tell** protocol or TMS protocol is used to set or change the truth value. **joshua:predication-truth-value** should only be used in modeling methods that implement those protocols.

Related Topics:

- **joshua:tell**
- **joshua:ask**
- **joshua:*true***
- **joshua:*false***
- **joshua:*unknown***
- **joshua:*contradictory***
- **joshua:truth-value**

See the section "Truth Values" in *User's Guide to Basic Joshua*.

joshua:prefetch-forward-rule-matches *predication context continuation* *Function*

predication	The pattern to be matched.
context	The entire *if*-part of the rule. Useful in advanced modeling applications. The default implementation ignores this argument, but rule compilation, where the way to compile one trigger depends on what other triggers are present, uses the context.
continuation	A function passed in which can determine whether a new rule trigger is necessary.

Takes a predication, a context, and a continuation and applies the continuation to all database predications that match the predication argument, without regard to truth value.

Its general use is for when rules are defined after some facts have already been entered into the database with **joshua:tell**. Newly installed rules may wish to trigger from those facts.

It is fairly rare that a user, even doing modelling, will need to define this method. The default definition, which may be inherited from **joshua:default-protocol-implementation-model**, simply uses the predication's **joshua:ask-data** method; the user will only need to define **joshua:prefetch-forward-rule-matches** if they do not define an **joshua:ask-data** method.

joshua:print-query *backward-support* &optional *(stream* ***standard-output****)* *Function*

A convenience function for use in an **joshua:ask** continuation. **joshua:print-query** displays the **joshua:ask** query with its variables instantiated.

backward-support The backward support supplied to the **joshua:ask** continuation.

stream A stream to which to output the information. Defaults to ***standard-output***.

Examples:

```
(define-predicate type-of (object type))

(tell [type-of Iliad epic])

(ask [type-of ?book epic] #'print-query)
[TYPE-OF ILIAD EPIC]
```

If you want to use the instantiated query in ways other than printing it, extract it yourself using the accessor function **joshua:ask-query**.

Related Functions:

joshua:ask
joshua:graph-query-results
joshua:print-query-results
joshua:say-query

See the section "Querying the Database" in *User's Guide to Basic Joshua*.

joshua:print-query-results *backward-support* &key *(:stream* ***standard-output****)* *(:printer* **#'prin1***)* *Function*

A convenience function for use in an **joshua:ask** continuation. **joshua:print-query-results** displays and interprets the support information in the **joshua:ask** continuation argument, *backward-support*; that is, it tells you what queries succeeded, and why.

backward-support A list containing the satisfied query and information about its support.

stream A stream to which to output the information. Default is ***standard-output***.

printer A function of two arguments, like **prin1**, that is used to print elements of the support. **prin1** is the default, but another reasonable value to give is **joshua:say**.

Use **joshua:graph-query-results** to see a graph of the information provided by **joshua:print-query-results**.

The accessor function **joshua:ask-derivation** extracts the support portion of *backward-support* but does not interpret the information.

For comparison, we use the same examples to illustrate all three functions.

Examples:

The first example shows a query satisfied by database lookup. Both the instantiated query and its support (here the matching database predication) are printed.

```
(define-predicate type-of (object type))

(tell [type-of Iliad epic])

(ask [type-of ?book epic] #'print-query-results)
[TYPE-OF ILIAD EPIC] succeeded: [TYPE-OF ILIAD EPIC] was TRUE in the database
```

The next example shows the support for a query that is satisfied from rules. We have a rule, dessert?, that determines if a given food is a dessert. Each of this rule's subgoals is derived from other rules. Here are the definitions.

```
(define-predicate edible (object))
(define-predicate is-food (object))
(define-predicate contains (object substance))
(define-predicate sweet (object))

(defrule food? (:backward)
  if [edible  ?object]
  then [is-food  ?object])

(defrule sweet? (:backward)
  if [or [contains ?object chocolate]
         [contains ?object sugar]
         [contains ?object honey]]
  then [sweet ?object])

(defrule dessert? (:backward)
  if [and [is-food ?object]
          [sweet ?object]]
  then [type-of ?object dessert])

;tell some sticky facts
(tell [edible chocolate-coated-ants])
(tell [contains chocolate-coated-ants honey])
```

Now we **joshua:ask** what foods qualify as desserts and why. A single food, chocolate-covered-ants, succeeded. The display shows the instantiated query, explaining why it succeeded: support is traced backward from rule dessert? that satisfied the query, through the support used to satisfy parts of the rule body.

```
  (ask [type-of ?object dessert] #'print-query-results)
[TYPE-OF CHOCOLATE-COATED-ANTS DESSERT] succeeded
  It was derived from rule DESSERT?
  [IS-FOOD CHOCOLATE-COATED-ANTS] succeeded
    It was derived from rule FOOD?
    [EDIBLE CHOCOLATE-COATED-ANTS] succeeded
      [EDIBLE CHOCOLATE-COATED-ANTS] was true in the database
  [SWEET CHOCOLATE-COATED-ANTS] succeeded
    It was derived from rule SWEET?
    [CONTAINS CHOCOLATE-COATED-ANTS HONEY] succeeded
      [CONTAINS CHOCOLATE-COATED-ANTS HONEY] was true in the database
```

Related Functions:

joshua:ask
joshua:graph-query-results
joshua:print-query
joshua:say-query

See the section "Querying the Database" in *User's Guide to Basic Joshua*. See the section "Explaining Backward Chaining Support" in *User's Guide to Basic Joshua*.

joshua:provable *proposition* *Joshua Predicate*

Checks if *proposition* is known to be **joshua:*true***, (or if it is known to be **joshua:*false***, if `[not ...]` is wrapped around it.)

This is a modal operator. `[provable ...]` and `[not [provable ...]]` correspond to the "box" and "diamond" operators of some modal logics.

proposition A Joshua predication pattern to match.

```
The query:        (ask [provable [foo ?x]] #' ...)
Succeeds when:    [foo ?x] would succeed

The query:        (ask [provable [not [foo ?x]] #' ...)
Succeeds when:    [not [foo ?x]] would succeed
```

If successful, **joshua:provable** calls the continuation on the instantiated query.

Examples:

Let's define a predicate, `shape-of`, **joshua:tell** some statements about the shape of objects, and then display the database.

```
(define-predicate shape-of (object shape))

(tell [and [shape-of door oval]
           [not [shape-of leaf pointed]]])
[AND [SHAPE-OF DOOR OVAL] [NOT [SHAPE-OF LEAF POINTED]]]
```

```
  Show Joshua Database
True things
  [SHAPE-OF DOOR OVAL]
False things
  [SHAPE-OF LEAF POINTED]]
```

Now we can check which statements about shapes are **joshua:*true***, and which are **joshua:*false***.

```
;; Check if the proposition is  joshua:*true*
(ask [provable [shape-of door oval]] #'print-query)
[PROVABLE [SHAPE-OF DOOR OVAL]]

;; Comparing provable to known
(ask [provable [shape-of leaf pointed]] #'print-query)
  ;this fails
(ask [known [shape-of leaf pointed]] #'print-query)
[KNOWN [SHAPE-OF LEAF POINTED]]

;; Check if the proposition is joshua:*false*
  (ask [provable [not [shape-of leaf pointed]]] #'print-query)
[PROVABLE [NOT [SHAPE-OF LEAF POINTED]]]

  (ask [provable [not [shape-of ?object ?shape]]] #'print-query)
[PROVABLE [NOT [SHAPE-OF LEAF POINTED]]]

;; Comparing provable to known
(ask [provable [not [shape-of door oval]]] #'print-query)
  ;this fails
(ask [known [not [shape-of door oval]]] #'print-query)
[KNOWN [NOT [SHAPE-OF DOOR OVAL]]]
```

It is more interesting to ask if something is *not* provable.

```
The query:        (ask [not [provable [foo ?x]]] #' ...)
Succeeds when:    [foo ?x] would have failed
```

```
;; Check if we don't know the proposition to be joshua:*true*
  (ask [not [provable [shape-of starfish round]]] #'print-query)
[not [PROVABLE [SHAPE-OF STARFISH ROUND]]]

;; Check if we don't know the proposition to be joshua:*false*
  (ask [not [provable [not [shape-of hill conical]]]] #'print-query)
[not [PROVABLE [NOT [SHAPE-OF HILL CONICAL]]]]
```

joshua:provable can also be used in backward rules.

Related Predicate:

joshua:known

Reset Joshua Tracing Command

Resets the tracing options to the original defaults.

Type of Tracing Which type of tracing to reset. The possible types are forward rules, backward rules, predications, TMS operations and All.

:Include events Whether to reset the traced and stepped events for the *Type of Tracing* as well.

The Reset Joshua Tracing command sets the Joshua tracing options back to their initial defaults. This command is useful if you have been selectively tracing rules or predications and would like to go back to tracing all rules or all predications. The *Include events* option comes in handy when you have been tracing or stepping particular events and would like to go back to just tracing the default events. This command does not disable or enable tracing, it just affects which things are traced.

Related Commands:

"Enable Joshua Tracing Command"
"Disable Joshua Tracing Command"

joshua:remove-action *slot-or-path* &optional *(name* **:action***)* *Generic Function*

This function is part of the Joshua object facility. It allows actions, which were added to slots using **joshua:add-action**, to be removed from those slots.

~\\say *predication* *Format Directive*

A **format** directive that makes it easy to combine the use of **joshua:say** with other kinds of formatted output. It takes one format-argument, the predication to be **joshua:say**'d to the output stream.

Examples:

```
(format t "~&The registry of deeds says that ~\\say\\."
        [frobozz Prospero 1616 remote-island])
```

This would print the following sentence:

```
The registry of deeds says that PROSPERO was an owner of a frobozz
in 1616 at REMOTE-ISLAND.
```

You can also use **~\\say** in other places **format** strings are used, for instance **prompt-and-accept**:

```
(prompt-and-accept 'integer "For what values of ~S is it true that ~\\say\\?"
                   ?x [Riemann-zeta 3 ?x])
```

Related Functions:

joshua:say

See the section "Formatting Predications: the SAY Method" in *User's Guide to Basic Joshua*.

joshua:say *predication* &optional *(stream* ***standard-output****)* *Function*

Prints out *predication* on *stream*, possibly in a way other than **prin1** would. This is good for printing the meaning of a predication in natural language, as opposed to the predicate calculus notation in which programs are written. However, you needn't restrict your thinking about **joshua:say** to just natural language. For example, **joshua:say** could present a predication as a piece of graphics; see examples below. Judicious use of **joshua:say** methods can make it easier to generate user interfaces.

It usually doesn't matter what value the implementations of **joshua:say** return, since **joshua:say** is usually done for side-effect. The exception is that if *stream* is explicitly supplied as **nil,** the implementations should do what **format** would do, that is, return a string if possible. (Graphical **joshua:say** methods can't do this.)

Examples:

```
(define-predicate frobozz (who when where) ()
 :destructure-into-instance-variables)

(define-predicate-method (say frobozz) (&optional (stream *standard-output*))
  (format stream "~S was an owner of a frobozz in ~S at ~S." who when where))

(say [frobozz Prospero 1616 remote-island])
```

prints the sentence:

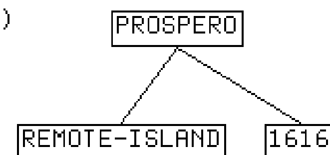
```
PROSPERO was an owner of a frobozz in 1616 at REMOTE-ISLAND.
```

An example using graphics would be:

```
(define-predicate-method (say frobozz) (&optional (stream *standard-output*))
  (dw:with-output-as-presentation
      (:stream stream :object self :type (type-of self))
    (format-graph-from-root (list who (list where) (list when))
                            #'(lambda (x s) (prin1 (car x) s))
                            #'cdr
                            :stream stream)))
```

The **joshua:say** method now draws a graph representing Prospero's relationship to his property and the time at which he owned it.

```
⇨ (say [frobozz Prospero 1616 remote-island])
```




```
#<DW::DISPLAYED-PRESENTATION [FROBOZZ PRO... JU::FROBOZZ 521636605>
```

Related Functions:

"~\\Say\\"

See the section "Formatting Predications: the SAY Method" in *User's Guide to Basic Joshua*.

joshua:say &optional *(stream* ***standard-output****)* of **joshua:predication** *Method*

The default implementation of **joshua:say**; It just prints *predication* in the same way **prin1** would, that is, using the bracket syntax. Its purpose is to make sure all predications support the **say** operation, even if only in a trivial fashion.

joshua:say-query *backward-support* &optional *(stream* ***standard-output****)* *Function*

A convenience function for use in an **joshua:ask** continuation. **joshua:say-query** displays the instantiated query using a user-defined **joshua:say** method if available, or the default **joshua:say** method. The latter simply prints the instantiated query.

backward-support The support supplied to the **joshua:ask** continuation.

stream A stream to which to output the information. The default is ***standard-output***.

Examples:

```
;; say-query with default say method
(define-predicate loves (person object))

(tell [loves Bob chocolate])

(ask [loves Bob ?x] #'say-query)
 [LOVES BOB CHOCOLATE]


;; say-query with user-defined say method
(define-predicate type-of (object type))

(define-predicate-method (say type-of) (&optional (stream *standard-output*))
  (with-statement-destructured (object type) ()
      (format stream
              "~% The ~A is an example of the ~A literary form." object type)))

(tell [type-of Iliad epic])
[TYPE-OF ILIAD EPIC]

(ask [type-of ?book epic] #'say-query)
 The ILIAD is an example of the EPIC literary form.
```

To use the instantiated query in some other way rather than **joshua:say**ing it, extract it from the continuation argument using the accessor function **joshua:ask-query**, and interpret the information.

Related Functions:

joshua:ask
joshua:graph-query-results
joshua:print-query
joshua:print-query-results

See the section "Querying the Database" in *User's Guide to Basic Joshua*.

Show Joshua Database Command

Displays the contents of the Database, or a subset of the contents matching a certain pattern.

matching pattern Specifies the predication patterns to display. The default is the entire database.

The display groups predications under the headings True and False, for predications with a truth value of **joshua:*true*** and **joshua:*false***, respectively.

When specifying a pattern you can further limit the display to patterns with either truth value.

Examples:

```
  Show Joshua Database (matching pattern [default All]) All
True things                  ; indication of truth value is in the heading
  [DREAMS-IN SPANISH LUCINDA]         [NATIVE-SPEAKER-OF SPANISH LUCINDA]
  [DREAMS-IN SUMERIAN DR-PARCHMENT]  [NATIVE-SPEAKER-OF GERMAN DR-PARCHMENT]
  [COUNTS-IN SPANISH LUCINDA]
False things
  [COUNTS-IN GERMAN HENRY]   ; indication of truth value is in the heading

  Show Joshua Database (matching pattern [default All]) [dreams-in ?x ?y]
       (opposite truth-value too? [default Yes]) Yes
True things
  [DREAMS-IN SPANISH LUCINDA]
  [DREAMS-IN SUMERIAN DR-PARCHMENT]
False things
  None
```

See the section "Entering and Displaying Predications in the Database" in *User's Guide to Basic Joshua*.

Show Joshua Predicates Command

Shows the currently defined Joshua predicates.

:*Include Models* Whether to include predicates that are used as base flavors for building other predicates in the output.

:*Matching* Show only predicates whose names contain a substring or substrings.

:*Output Destination* Where to display the information.

:*Packages* Only show predicates in the specified package or packages. Supply a value of All to see all the currently defined Joshua predicates. Unless you otherwise specify the package, you see only the predicates defined in the current package.

:*Search Inherited Symbols*
Whether to include predicates that are inherited by the packages specified in :*Packages*.

:*System* Show only the predicates that are defined in a particular system.

The Show Joshua Predicates command provides a convenient tool for browsing through all the predicates defined in the current world. The output is a table of predicate names and arguments. There are a number of mouse behaviors defined for the predicate names that this command displays. These can be seen by mousing right on the name.

```
Show Joshua Predicates :Packages TME
TME:ABNORMAL (WHO FOR-WHAT)   TME:LOVES (LOVER LOVEE)
TME:BIRD (BOID)               TME:ONE-PER-ROW-OR-COL (R-OR-C INDEX)
TME:FLY (BOID)                TME:PENGUIN (BOID)
IS-EXAMPLE-OF (NAME TYPE)     PROVABLE (PROPOSITION)
TME:JEALOUS (WHO)             TME:QUEEN (ROW COL)
TME:KILLS (KILLER VICTIM)     TME:TRAGEDY (EVENT)
KNOWN (PROPOSITION)
```

Related Commands:

"Show Joshua Rules Command"
"Show Joshua Tracing Command"

Show Joshua Rules Command

Displays the currently defined rules.

:*Triggered By* Show rules with one or more triggers that unify with the specified predication.

:*Matching* Show rules with names containing one or more substrings.

:*Output Destination* Where to display the output from this command.

:*Packages* Show the rules defined in which package or packages. This defaults to the current package.

:*Search Inherited Symbols*
Include rules that are inherited by *Packages*.

:*System* Show only the rules defined in a particular system.

:*Type* Show only backward or forward rules. By default the command shows both backward and forward rules.

The Show Joshua Rules command provides a tool for browsing through all the Joshua rules. It displays a table of all the rules satisfying the given arguments. Mousing middle on a rule name displays the most recent definition of that rule.

Example:

```
Show Joshua Rules :Triggered By [tme:loves ? ?] :Packages All

Forward Rules:
JEALOUSY        LOVE-IN-IDLENESS ONLY-ONE-LOVE QUALITY-NOT-QUANTITY
UNREQUITED-LOVE
```

The above example lists all of the rules that could be triggered by a predication of the form `[tme:loves ? ?]`.

Related Commands:

"Show Joshua Predicates Command"
"Show Joshua Tracing Command"

Show Joshua Tracing Command

Shows information about Joshua tracing.

Type of Tracing Which type of tracing to describe. It can be one of forward rules, backward rules, predications, TMS operations, or all.

:*Output Destination* Where to display the output from this command.

The Show Joshua Tracing command describes the current state of Joshua tracing, saying whether each *Type of Tracing* is on or off. For each active *Type of Tracing*, Show Joshua Tracing prints out information about the current options and traced events.

Example:

```
Show Joshua Tracing (type of tracing) All
```

```
Foward Chaining tracing is on
   Tracing all forward rules triggered by a predication matching:
      [TME:LOVES DEMETRIUS HERMIA]
   Traced events: Fire and Queue

TMS tracing is off

Predication tracing is on
   Tracing predications of flavor: LTMS:LTMS-PREDICATE-MODEL
   Traced events: Ask and Tell

Backward Chaining tracing is off
```

Related Commands:

"Show Joshua Rules Command"
"Show Joshua Predicates Command"

Show Rule Definition Command

Shows the latest definition of a Joshua rule.

Rule Show the definition of which rule or rules.

:*Load* This argument controls the behavior of the command when the desired rule definition is not currently in an editor buffer. If you enter Yes, the command loads the definition into an editor buffer. If you enter No, it does not. The value of *Load* defaults to Query, meaning the command should ask you before loading any file into the editor.

:*Output Destination* Where to display the output from this command.

The Show Rule Definition command allows you to see the definition of a Joshua rule in a Lisp Listener without having to enter the editor. When the rule definition can be found in the editor the command displays the latest version. Otherwise, depending on the value of *Load*, the command offers to read in the latest definition from the file containing the rule definition.

Example:

```
Show Rule Definition JEALOUSY
Rule Jealousy:
(defrule jealousy (:forward :importance 3)
  IF [and [jealous ?x]
          [loves ?x ?y]
          [loves ?z ?y]
          (different-objects ?x ?z)]
  THEN [kills ?x ?z])
```

joshua:slot-current-predication *slot* *Generic Function*

This function is part of the Joshua object facility. It finds the predication expressing the current object-attribute-value triple represented by the slot.

joshua:slot-current-value *slot* *Generic Function*

This function is part of the Joshua object facility. It finds the current value of a slot.

Note that the meaning of this value may be dependent upon the type of a slot: for instance in the case of set-valued slots, the value may be a list representing the set.

joshua:slot-my-object *slot* *Generic Function*

This function is part of the Joshua object facility. Given a slot, it finds the object that owns that slot.

joshua:slot-value-mixin *Flavor*

This flavor-mixin is part of the Joshua object facility. It may be used to add slot-value behaviour, like that of the default slot-value predicate **joshua:value-of**, to predicate models defined by the user.

joshua:succeed &optional *support* *Function*

Joshua is a success-continuation-passing language. In most places, calling the continuation means "go ahead with the rest of the computation". Based on context, the form **joshua:succeed** finds the continuation and calls it accordingly.

You can use **joshua:succeed** within Lisp code embedded in:

- The *if*-part of rules (in Lisp code in forward rules, and in multiply-succeeding Lisp forms of backward rules)
- The body of a **joshua:defquestion**

It makes no sense to call **joshua:succeed** elsewhere.

The optional *support* argument allows the Lisp code to specify the derivation information for the query.

Example:

```
(define-predicate good-to-read (book))

(defparameter *books* '(decameron canterbury-tales gargantua-and-pantagruel
                                  tom-jones catch-22))

(defrule reading-list (:backward)
  if (typecase ?candidate-book
       (unbound-logic-variable
         (loop for book in *books*
               doing (with-unification
                       (unify ?candidate-book book)
                       (succeed 'Humor-101-reading-list))))
       (otherwise
         (when (member ?candidate-book *books*)
           (succeed (succeed 'Humor-101-reading-list)))))
  then [good-to-read ?candidate-book])
```

```
(ask [good-to-read ?x] #'print-query-results)
[GOOD-TO-READ DECAMERON] succeeded
  It was derived from rule READING-LIST
  HUMOR-101-READING-LIST
[GOOD-TO-READ CANTERBURY-TALES] succeeded
  It was derived from rule READING-LIST
  HUMOR-101-READING-LIST
[GOOD-TO-READ GARGANTUA-AND-PANTAGRUEL] succeeded
  It was derived from rule READING-LIST
  HUMOR-101-READING-LIST
[GOOD-TO-READ TOM-JONES] succeeded
  It was derived from rule READING-LIST
  HUMOR-101-READING-LIST
[GOOD-TO-READ CATCH-22] succeeded
  It was derived from rule READING-LIST
  HUMOR-101-READING-LIST
```

Related Functions:

joshua:unify
joshua:with-unification

joshua:support *database-predication* &optional *filter* *Function*

Examines the TMS justification structures currently supporting belief in *database-predication*, tracing them back to primitively justified predications (i.e. to those whose support does not depend on any other predications). Returns a list of the primitive support (assumptions and premises). *Filter*, if provided, is a predicate to be applied to the support. Only those elements of the primitive support which satisfy the predicate are collected.

database-predication A predication object that is in the database. Must be the actual database object, and not a copy of it.

filter If *filter* is not supplied the value default to **nil** which means that all the primitive support should be collected and returned. Otherwise, *filter* should be a function of one argument that returns non-**nil** on the support you want. (For example, you might want to look at just the assumption support of *database-predication*.) When the *database-predication* argument is based on a TMS, this function is passed a justification as its argument. It may examine the justification using **joshua:destructure-justification**.

Examples:

Prospero, curious about his daughter's relationship with Caliban, might do:

```
(ask [is-friend-of Miranda ?]
     #'(lambda (backward-support)
         (format t "~&The support for ~S is ~S."
                   (ask-database-predication backward-support)
                   (support (ask-database-predication backward-support))))
     :do-backward-rules nil)
```

If he wanted to see just the assumptions underlying it, he would do:

```
(ask [is-friend-of Miranda ?]
     #'(lambda (backward-support)
         (format t "~&The support for ~S is ~S."
                   (ask-database-predication backward-support)
                   (support (ask-database-predication backward-support)
                            #'(lambda (justification)
                                (multiple-value-bind (ignore mnemonic)
                                   (eq mnemonic :assumption))))))
     :do-backward-rules nil)
```

See the section "The Truth Maintenance Protocol", page 54.

joshua:support &optional *filter* of **joshua:default-protocol-implementation-model** *Method*

This is the default implementation the the **joshua:support** protocol function. It returns **nil**. Predications that do provide a TMS should be based on **joshua:basic-tms-mixin**, which defines a **joshua:support** method that provides real information.

joshua:support &optional *filter* of **joshua:basic-tms-mixin** *Method*

This is the default implementation of the **joshua:support** protocol function for all models that implement a TMS. Any TMS implementation may use this method simply by mixing in the **joshua:basic-tms-mixin** flavor. If the TMS implementor needs to provide functionality not provided by this method, that can be done by providing a method for **joshua:support** with the model that implements the new TMS. Most users will never need to know about this.

joshua:tell *predication* &key *:justification* *Function*

Puts a predication into the virtual database.

Note: joshua:tell is a macro, and as such it cannot be used as an argument to the function **funcall**.

predication should be thought of as a pattern argument, not as the actual data in the database. If something already exists in the database that is a **joshua:variant** of *predication*, the returned (canonical) value will not be **eq** to *predication*. Thus **joshua:tell** serves as an interner for *predication*, that is, it gives you the canonical copy in the database, creating it if necessary.

If *predication* is not already in the database, the returned values are *predication* and the symbol **t**.

If something already exists in the database that is a **joshua:variant** of *predication*, *predication* is not put into the database, since that would be duplication. Instead, the canonical version found in the database is returned, along with the symbol **nil**.

Justification can be one of the following:

- **nil**, in which case a default justification is used. If the **joshua:tell** occurs outside a rule, the default justification is **:premise**. If the **joshua:tell** is inside a rule, the default justification includes the rule name and the current support set.

- **A symbol.** A justification which is a symbol means that the truth-value of *predication* does not depend on that of any other predication; we say that *predication* has a *primitive justification*, in such a case. One primitive justification is specially treated by the LTMS provide with Joshua, namely **:premise**. **:premise** justifications will never be removed by the LTMS without querying the user. Other primitive justifications are treated as assumptions that can be removed by the LTMS if necessary to resolve a contradiction.

- **A List of Four fields.** These are identical to the arguments to the Joshua protocol function **joshua:justify**, namely a *mnemonic*, *true-support*, *false-support* and *unknown-support*. These fields are used (or discarded) by whatever TMS is present.

The database into which *predication* is put depends on the data model of its predicate. The default is the discrimination net.

Examples:

```
(tell [is-magician Prospero])
(tell [not [is-magician Caliban])
(tell [is-daughter-of Miranda Prospero])
(tell [is-servant-of Caliban Prospero] :justification :premise)
(tell [is-friend-of Miranda Caliban] :justification :assumption)
  ;later retracted!
(tell '[is-exiled-from Prospero ,(find-exile-country 'Prospero)])
```

Note:

Chances are that you seldom want to define a method that takes over the entire functionality of **joshua:tell.** It's more likely that you would want to define a method for one of the generic functions it calls, such as **joshua:insert**, **joshua:justify,** or **joshua:map-over-forward-rule-triggers**.

Related Functions:

joshua:untell
joshua:clear
joshua:ask
joshua:justify

See the section "Entering and Displaying Predications in the Database" in *User's Guide to Basic Joshua*.

See the section "The Joshua Database Protocol", page 8.

See the section "Customizing the Data Index", page 81.

See the section "Truth Maintenance Facilities", page 53.

joshua:tms-bits *predication* *Generic Function*

predication Any predication

Predications contain a word of flag bits for use by internals of the system. Several of these flags are reserved for use by TMS implementors. This function retrieves these bits from a predication. The meaning of the field of bits returned is defined by the specific TMS.

See the section "The Truth Maintenance Protocol", page 54.

joshua:tms-contradiction *Flavor*

A TMS initiates backtracking in Joshua by signalling a condition which is based upon the **joshua:tms-contradiction** flavor. Instances of this flavor include several instance variables containing information useful to the default condition handler or to any handler which attempts to conduct intelligent backtracking. There are accessors defined for each of these instance variables. Joshua provides a default handler for this condition which examines the primitive support underlying the contradiction.

If the primitive support of the contradiction contains non-premises, then the default handler offers the user the opportunity to retract one or more of these. This will continue until the contradiction is resolved. If the primitive support of the contradiction contains a single non-premise, then the default handler automatically retracts that predication without interacting with the user.

The default handler can be overridden by using **condition-bind** to bind the **joshua:tms-contradiction** condition.

If the primitive support contains only premises then the situation is regarded as more serious since premises are not supposed to be retraced by a TMS automatically. In this case the default handler signals a *hard contradiction* condition. See the generic function **joshua:tms-contradiction-hard-contradiction-flavor**, page 237. The hard contradiction condition is handled by a default handler which offers the user the opportunity to retract a member of the premise support of the contradiction.

This default handler can be overridden by using **condition-bind** to bind the **joshua:tms-hard-contradiction** condition.

Specific TMS's may provide their own contradiction conditions by defining a flavor which mixes in the **joshua:tms-contradiction** flavor.

Users may also develop a more elaborate contradiction signalling mechanism by defining conditions of their own which mix in the **joshua:tms-contradiction** flavor. Specific condition handlers for these conditions may also be defined, allowing a fine-grained control of the backtracking process. See the section "Signalling Conditions" in *Symbolics Common Lisp Programming Constructs*. See the section "Signalling Contradictions and Managing Backtracking", page 57.

joshua:tms-contradiction-contradictory-predication *tms-contradiction* *Generic Function*

tms-contradiction A condition object built on the flavor **joshua:tms-contradiction**.

A TMS initiates backtracking in Joshua by signalling a condition which is based upon the **joshua:tms-contradiction** flavor. Instances of this flavor include several instance variables containing information useful to the default condition handler or to any handler which attempts to conduct intelligent backtracking. There are accessors defined for each of these instance variables. This generic function accesses an instance variable which contains the predication which initiated backtracking. A TMS may choose not to provide any information in this field if it detects the contradiction a part of a global process which does not allow the contradiction to be isolated to an individual predication.

See the section "Signalling Contradictions and Managing Backtracking", page 57.

joshua:tms-contradiction-hard-contradiction-flavor *tms-contradiction* *Generic Function*

tms-contradiction A condition object built on the flavor **joshua:tms-contradiction**.

A TMS initiates backtracking in Joshua by signalling a condition which is based upon the **joshua:tms-contradiction** flavor. Instances of this flavor include several instance variables containing information useful to the default condition handler or to any handler which attempts to conduct intelligent backtracking. There are accessors defined for each of these instance variables. This generic function returns the name of the hard contradiction flavor associated with *tms-contradiction*. This is the condition which should be signalled if the current contradiction includes only premises in its primitive support. See the flavor **joshua:tms-hard-contradiction**, page 239. See the section "Signalling Contradictions and Managing Backtracking", page 57.

joshua:tms-contradiction-justification *tms-contradiction* *Generic Function*

tms-contradiction A condition object built on the flavor **joshua:tms-contradiction**.

A TMS initiates backtracking in Joshua by signalling a condition which is based upon the **joshua:tms-contradiction** flavor. Instances of this flavor include several instance variables containing information useful to the default condition handler or to any handler which attempts to conduct intelligent backtracking. There are accessors defined for each of these instance variables.

This generic function accesses an instance variable which contains the justification that initiated backtracking. If a specific predication initiated backtracking, then this function returns the justification of that predication. See the generic function **joshua:tms-contradiction-contradictory-predication**, page 237. However, a TMS (e.g. Joshua's LTMS) may detect the contradiction as part of a global process which localizes the contradiction not to a predication but to a justification which cannot be satisfied. In such a case, this generic function returns the unsatisfiable justification, but the generic function **joshua:tms-contradiction-contradictory-predication** returns **nil**.

See the section "Signalling Contradictions and Managing Backtracking", page 57.

joshua:tms-contradiction-non-premises *tms-contradiction* *Generic Function*

tms-contradiction A condition object built on the flavor **joshua:tms-contradiction**.

A TMS initiates backtracking in Joshua by signalling a condition which is based upon the **joshua:tms-contradiction** flavor. Instances of this flavor include several instance variables containing information useful to the default condition handler or to any handler which attempts to conduct intelligent backtracking. There are accessors defined for each of these instance variables. This generic function accesses an instance variable which contains a subset of the primitive-support underlying a contradiction. The subset includes all elements of the primitive-support which the TMS regards as retractable, that is, everything except the premises.

See the section "Signalling Contradictions and Managing Backtracking", page 57.

joshua:tms-contradiction-premises *tms-contradiction* *Generic Function*

tms-contradiction A condition object built on the flavor **joshua:tms-contradiction**.

A TMS initiates backtracking in Joshua by signalling a condition which is based upon the **joshua:tms-contradiction** flavor. Instances of this flavor in-

clude several instance variables containing information useful to the default condition handler or to any handler which attempts to conduct intelligent backtracking. There are accessors defined for each of these instance variables. This generic function accesses an instance variable which contains a subset of the primitive-support underlying a contradiction. The subset includes all elements of the primitive-support which the TMS regards as not retractable, that is, the premises.

See the section "Signalling Contradictions and Managing Backtracking", page 57.

joshua:tms-contradiction-support *tms-contradiction* *Generic Function*

tms-contradiction A condition object built on the flavor **joshua:tms-contradiction**.

A TMS initiates backtracking in Joshua by signalling a condition which is based upon the **joshua:tms-contradiction** flavor. Instances of this flavor include several instance variables containing information useful to the default condition handler or to any handler which attempts to conduct intelligent backtracking. There are accessors defined for each of these instance variables. This generic function accesses the instance variable which contains all the primitive support underlying a contradiction.

See the section "Signalling Contradictions and Managing Backtracking", page 57.

joshua:tms-hard-contradiction *Flavor*

This flavor is the base flavor upon which to build condition objects for *Hard Contradictions*. A hard contradiction is signalled when there is a contradiction whose primitive support includes only premises (i.e. primitive support which the TMS is not free to retract automatically).

A TMS initiates backtracking in Joshua by signalling a condition which is based upon the **joshua:tms-contradiction** flavor. Instances of this flavor include several instance variables containing information useful to the default condition handler or to any handler which attempts to conduct intelligent backtracking. There are accessors defined for each of these instance variables.

A hard contradiction condition is not normally signalled directly by a TMS or a user's program. They should instead signal a condition built upon **joshua:tms-contradiction**. The default handler for this condition will, in turn, signal a hard contradiction if there are only premises in the primitive support. To do this, the handler needs to know the name of the hard contradiction flavor corresponding to the contradiction condition signalled; this information is provided by the generic function **ju::hard-contradiction-flavor** which must be implemented by any flavor built upon **joshua:tms-contradiction**.

See the section "Signalling Contradictions and Managing Backtracking", page 57.

joshua:*true* *Variable*

A named constant used by Joshua to denote a truth value of true. You can compare truth values using **eql**.

Related Topics:

joshua:*false*
joshua:*unknown*
joshua:*contradictory*
joshua:truth-value
joshua:predication-truth-value

See the section "Truth Values" in *User's Guide to Basic Joshua*.

joshua:truth-value *Presentation Type*

This type provides a convenient way to accept and present truth values. It will parse the truth-value name and return the integer value for that truth-value. When presenting truth-values it will present the numeric truth value as one of true, false, unknown, or contradictory.

Examples:

```
(accept 'truth-value)
Enter a truth value: true
1
TRUTH-VALUE

(present 2 'truth-value)false
#<DW::DISPLAYED-PRESENTATION 2 JOSHUA:TRUTH-VALUE 513174521>
```

Related Function:

joshua:predication-truth-value

joshua:type-of-mixin *Flavor*

This flavor-mixin is part of the Joshua object facility. It may be used to add object-type behaviour, like that of the default object-type predicate **joshua:object-type-of**, to predicate models defined by the user.

joshua:type-of-mixin inherits from **joshua:tell-error-model** and **joshua:ask-data-only-mixin**.

joshua:undefine-predicate *name* *Macro*

"Undoes" a predicate definition. Predications built with this definition remain in the world, but an attempt to do almost anything to them results in an error.

Example:

```
(define-predicate fruit (a-fruit))
(undefine-predicate 'fruit)
```

You can perform the same operation from the Zmacs editor. Place your cursor on the predicate definition to be removed and use the command m-X Kill Definition. The system asks for confirmation in the minibuffer; then it offers you the options of removing the definition from the editor buffer itself, and of inserting the **joshua:undefine-predicate** command into the editor buffer.

Example:

1. Interaction During m-X Kill Definition

```
;;; -*- Mode: Joshua; Package: JOSHUA-USER; Syntax: Joshua; Vsp: 0 -*-
;;; Created 8/07/87 14:15:27 by Covo running on LADY-PEREGRINE at SCR

(define-predicate needs-a-vacation (person))

Extended command:
Remove predicate NEEDS-A-VACATION from the current world? (Y or N) Yes.
Remove predicate NEEDS-A-VACATION from the editor buffer? (Y or N) No.
Insert form to kill predicate NEEDS-A-VACATION into the editor buffer? (Y or N)
```

2. Zmacs Buffer After Completion of m-X Kill Definition

```
;;; -*- Mode: Joshua; Package: JOSHUA-USER; Syntax: Joshua; Vsp: 0 -*-
;;; Created 8/07/87 14:15:27 by Covo running on LADY-PEREGRINE at SCRC.

(UNDEFINE-PREDICATE 'NEEDS-A-VACATION)
(define-predicate needs-a-vacation (person))

Zmacs (Joshua) questions-examples.lisp >sys>joshua>doc>examples Q: (15) *  [More below]
Predicate NEEDS-A-VACATION removed from the current world.
```

Related Functions:

> **joshua:define-predicate**
> "Zmacs Command: Kill Definition"

joshua:undefine-predicate-method *method-spec* *Function*

Removes the method defined for *method-spec* from the world.

method-spec A Joshua protocol method specifier of the form *(protocol-function flavor* &rest *options).*

The editor command m-X Kill Definition is an easy way to remove a predicate method for a method defined in an editor buffer.

Related function:

> **joshua:define-predicate-method**

joshua:undefine-predicate-model *name* *Function*

Removes the predicate named *name* from the world.

name The name of a predicate model.

The editor command m-X Kill Definition is an easy way to remove a predicate model for a model defined in an editor buffer.

Related function:

joshua:define-predicate-model

joshua:undefquestion *name* *Function*

Removes a question definition from the system.

name The name of the question

```
(define-predicate foo (something something-else))

(defquestion question1 (:backward) [foo 1 ?x])

  (ask [foo 1 2] #'print-query :do-questions t)
Is it true that "[FOO 1 2]"?  [default No]: Yes
[FOO 1 2]

  (undefquestion 'question1)
QUESTION1

  (ask [foo 1 2] #'print-query :do-questions t)
```

To kill a question definition from a Zmacs buffer, use the command m-X Kill Definition. For a sample interaction with the command: See the macro **joshua:undefine-predicate**, page 240.

Related Functions:

joshua:defquestion
"Zmacs Command: Kill Definition"

See the section "Asking the User Questions" in *User's Guide to Basic Joshua*.

joshua:undefrule *rule-name* *Function*

Removes a rule definition so that the rule cannot execute.

You can also remove a rule from a Zmacs buffer with m-X Kill Definition. For a sample interaction with the command: See the macro **joshua:undefine-predicate**, page 240.

rule-name The name of the rule to be removed.

Examples:

```
(defrule parched (:forward)
  if [condition-of plant-soil dry]
  then [needs plant-soil water])

(undefrule 'parched)
```

Modeling Note:

joshua:undefrule calls one of the generic functions **joshua:delete-forward-rule-trigger** or **joshua:delete-backward-rule-trigger** which removes the rule's trigger from its storage place, so that it is no longer found by the trigger locating and trigger mapping functions.

See the section "The Contract of the Trigger Deleting Functions", page 38.

Related Functions:

joshua:defrule
joshua:clear
"Clear Joshua Database Command"
"Zmacs Command: Kill Definition"

See the section "Rules and Inference" in *User's Guide to Basic Joshua*.

joshua:unify *object1 object2* *Function*

If *object1* and *object2* unify, does so, while side-effecting any logic variables for the duration of the unification.

object1	A pattern in Joshua, that is, a predication containing other predications, lists, symbols, numbers, or logic variables.
object2	Another pattern.

Pattern matching underlies the inferencing process. In forward chaining, Joshua matches rule trigger patterns with database predications. In backward chaining, it matches goals with database predications and with rule and question trigger patterns.

Two patterns containing no logic variables *match* if they are structurally equivalent (if they "look the same").

Two patterns containing logic variables *unify* when one can substitute values for the variables so that both patterns become structurally equivalent. The process of doing so is called *unification*.

joshua:unify is useful for assigning values to logic variables within Lisp code in rule bodies. If the expressions are unifiable, the appropriate substitutions are made and rule execution continues.

If the expressions are not unifiable, rule execution fails. "Fails" means that it throws to the nearest (dynamically) containing **joshua:with-unification** clause.

Always wrap the macro **joshua:with-unification** around **joshua:unify** (or calls to functions that call **joshua:unify**) to establish the scope within which the substitutions remain in effect.

The Joshua unifier does what is called an *occur check*, that is, prevents the formation of certain circular structures by refusing to unify a logic variable with a structure in which it occurs. For example, if you tried to unify `?x` with `[f ?x]`, you would get something whose printed representation would look (partially) like this:

```
[f [f [f [f [f [f ...
```

This is exactly the same thing that happens when you make certain conses point at themselves — you get circular lists.

To see how this might happen, consider example 3 below.

Examples:

```
Example 1:
(define-predicate yearly-salary (employee salary))
(define-predicate balance-due (person balance))
(define-predicate deny-credit (person))

(defrule test-1 (:forward)
  if [and [balance-due ?applicant ?balance]
          [yearly-salary ?applicant ?salary]
          (unify ?cash-flow (- ?salary ?balance))
          (≤ ?cash-flow ?balance)]
  then [and [deny-credit ?applicant]
            (format t "~% Sorry, ~S, your cash-flow of ~S is insufficient."
                    ?applicant ?cash-flow)])

(defun test-it ()
  (clear)
  (tell [yearly-salary Fred 20000])
  (tell [balance-due Fred 15000])
  (tell [yearly-salary George 200000])
  (tell [balance-due George 15000])
  'done-testing)
TEST-IT

(test-it)
Sorry, FRED, your cash flow of 5000 is insufficient.
DONE-TESTING
```

```
  Show Joshua Database
True things
  [BALANCE-DUE FRED 15000]
  [YEARLY-SALARY FRED 20000]
  [YEARLY-SALARY GEORGE 200000]
  [BALANCE-DUE GEORGE 15000]
  [DENY-CREDIT FRED]          ;Inference added to database
False things
  None


Example 2:
(with-unbound-logic-variables (x)
  (let ((p1 '[foo ,x])
        (p2 [foo 1]))
     (with-unification
       (unify p1 p2)
         ; If p1 and p2 don't unify, the next
         ;   expression is not executed
       (format t "~&The value of x is ~s." x))))
The value of x is 1.
NIL
```

Example 3 shows a case where the occur-check feature makes the unification fail.

```
Example 3:
(define-predicate f (arg))
(define-predicate g (arg1 arg2))

(defun test-occur ()
  (with-unbound-logic-variables (x y)
    (with-unification
      (unify '[g ,x ,x] '[g ,y [f ,y]])
      ;; if you get here, print Y and return
      (format t "~&You blew it.  Y is now circular: ~S" y)
      (return-from test-occur :loser))
    ;; if you got here, the unification failed
    :occur-check-forbids))

(test-occur)
:OCCUR-CHECK-FORBIDS
```

This function attempts to unify `[g ?x ?x]` with `[g ?y [f ?y]]`. If it unifies, the function prints an abusive message and returns the symbol `:loser`. If the unification fails, it returns the symbol `:occur-check-forbids`.

Let's follow the unification and see what happens:

- The predicates in both places are g, so the unifier goes on to inspect the arguments.
- The first argument on the left is ?x and the first on the right is ?y. The unifier unifies ?y and ?x, which we can write as the equation ?x = ?y.
- The next argument on the left is ?x and the next on the right is [f ?y]. Thus the unifier attempts to enforce the equation ?x = [f ?y].

We thus have the two equations ?x = ?y and ?x = [f ?y]. Combining them, we have the single equation ?y = [f ?y], whose only solution is to unify ?y to a structure containing itself, that is, a predication that structurally resembles a circular list: [f [f [f [f .... The unifier forbids this and fails. When the unifier fails, it throws to the nearest containing **joshua:with-unification**. Thus the function above returns :occur-check-forbids.

```
(test-occur) -> :occur-check-forbids
```

Why should Joshua attempt to avoid creating such circular structures, though? (The check does have a cost in performance, which is why most versions of Prolog won't do it.) The answer is that if it were permitted, certain incorrect inferences could be made. Here's an example. Suppose we have a predicate is-parent-of, which takes two people as arguments:

```
(define-predicate has-parent (kid parent))
```

This means that parent is a parent of kid. We can then make the (unsurprising) statement that every person has a parent:

```
∀x ∃y : has-parent(x, y)
```

or, in quantifier-free language,

```
[has-parent ?x (p ?x)]
```

where p is the Skolem function for the existential variable y. (You can think of it as a notation for finding the parent of its argument.)

Now try to unify the above statement with [has-parent ?z ?z]. In the absence of the occur check, we get the equations:

```
?z = ?x

and

?z = (p ?x)
```

(This would end up with ?x = (p ?x) = (p (p (p (p ...). Now substitute for the arguments in [p ?z ?z] using those equations, to get:

```
[has-parent (p ?x) ?x]
```

which is just the original statement with the arguments reversed. *This is unsound*. It is not justifiable to infer that has-parent is a symmetric predicate. (Indeed, it is not, since no one is his own parent!) Thus, to be sound, Joshua must forbid occur-check-type matches.

Related Functions:

> **joshua:with-unification**
> **joshua:succeed**

See the section "Pattern Matching in Joshua: Unification" in *User's Guide to Basic Joshua*.

joshua:uninsert *database-predication* *Generic Function*

joshua:uninsert removes a single *database-predication* that **joshua:insert** had previously stored in the database. **joshua:uninsert** is a subroutine of **joshua:untell**; other subroutines called by **joshua:uninsert** handle other aspects of removing up all vestiges of *database-predication* from the Joshua world.

Note that **joshua:uninsert** does not "search" for predications that match its argument as **joshua:ask** does. **joshua:uninsert** only removes its "argument" from the database, usually testing with **eq**.

See the section "The Joshua Database Protocol", page 8.

joshua:unjustify *database-predication* &optional *justification* *Generic Function*

Removes a justification from a predication in the database. For example, if you **joshua:tell** *predication* and then later change your mind about it, you can use **joshua:unjustify** to remove *justification* from the possible supports. This does not automatically remove all support for *database-predication*, as there might be other justifications for it as well.

database-predication A predication object that is in the database. Must be the actual database object, and not a copy of it.

justification Specifies the justification to be removed. If *justification* is not supplied, implementations of **joshua:unjustify** should default it to the justification currently being used to support *database-predication*.

In general, **joshua:unjustify** is useful only if *database-predication* is built on some model that supports the TMS protocol.

In the default (non-TMS) Joshua model, **joshua:unjustify** just sets the truth-value of its argument to **joshua:*unknown***.

Examples:

When Prospero is reconciled to his countrymen, he will cast the following spell:

```
(map-over-database-predications [is-exiled-from Prospero ?] #'unjustify)

(map-over-database-predications [is-exiled-from Miranda ?] #'unjustify)

(map-over-database-predications [is-friend-of Miranda Caliban] #'unjustify)
```

joshua:unjustify and **joshua:untell** work in similar fashion, but with very

different results. See the generic function **joshua:untell**, page 248. **joshua:unjustify** keeps the unjustified fact in the database. If the fact is later given again to **joshua:tell**, it is not considered as a new predication, but rather as a variant of an existing one, and no forward rules are run.

joshua:untell, on the other hand, actually removes the fact from the database, freeing up storage, and causing the database to lose previous knowledge of it; if the fact is later given to **joshua:tell** again, it is considered as a new fact, and forward rules are rerun.

Related Functions:

joshua:untell
joshua:uninsert

See the section "Revising Program Beliefs" in *User's Guide to Basic Joshua*.

See the section "Retracting Predications with **joshua:unjustify**" in *User's Guide to Basic Joshua*.

joshua:unjustify &optional *justification* of **ltms:ltms-mixin** *Method*

The **joshua:unjustify** method for the LTMS. It removes an LTMS format justification (i.e. a clause) from *predication*. *Justification* defaults to the current justification. See the theory of the LTMS for details.

joshua:*unknown* *Variable*

A named constant used by Joshua to denote a truth value of **joshua:*unknown***. You can compare truth values using **eql**.

A predication is **joshua:*unknown*** when there is no valid reason that supports it. The predication may or may not remain in the database, but is conceptually "not seen" until its truth value changes to **joshua:*true*** or **joshua:*false***.

Related Topics:

joshua:*true*
joshua:*false*
joshua:*contradictory*
joshua:truth-value
joshua:predication-truth-value

See the section "Truth Values" in *User's Guide to Basic Joshua*.

joshua:untell *database-predication* *Generic Function*

Removes a single predication from the database, clearing up storage space. (This function is a dual of **joshua:tell**, which *adds* a predication to the database.)

database-predication A predication. Must be the actual predication object that is in the database, not a copy of it.

joshua:untell first calls **joshua:unjustify** to make the fact no longer be-

lieved (**joshua:*unknown***), clears some internal caches, then calls **joshua:uninsert** to remove the fact from the database. The surgical properties of **joshua:untell** in actually removing the predication as opposed to only removing its justification have two effects:

1. Some storage may become garbage-collectible. This can lower the virtual-memory requirements of your program. Of course, you pay for it by doing the extra work of **joshua:uninsert**.

2. The predication is no longer in the database. This means that if you re-**joshua:tell** it, **joshua:tell** returns a second value of **T**, denoting it has never seen this predication before; in consequence, **joshua:tell** will also run forward rules. again.

(If, on the other hand, you merely **joshua:unjustify** the predication, then **joshua:tell** it once again, **joshua:tell** returns a second value of **nil**, denoting the predication already existed in the database; **joshua:tell** does not run forward rules when an existing predication is retold.) However, if a TMS is present, the consequences of running those rules will be brought back in.

Examples:

```
(define-predicate has-eye-color (creature color))

(tell [and [has-eye-color cat green]
           [has-eye-color rat black]])

  Show Joshua Database
True things
  [HAS-EYE-COLOR CAT GREEN]
  [HAS-EYE-COLOR RAT BLACK]
False things
  None

;; untell a predication by clicking left on it in the database display
  (untell [HAS-EYE-COLOR CAT GREEN])
NIL

  Show Joshua Database (matching pattern [default All]) All
True things
  [HAS-EYE-COLOR RAT BLACK]
False things
  None
```

```
;; untell using the predication object returned as the query support
  (ask [has-eye-color rat black]
     #'(lambda  (backward-support)
         (untell (ask-database-predication backward-support)))
     :do-backward-rules nil)

   Show Joshua Database (matching pattern [default All]) All
True things
  None
False things
   None
```

Note that in the last example above you probably should have used

```
(map-over-database-predications [has-eye-color rat black] #'untell)
```

Compare the following examples to see the difference between **joshua:untell** and **joshua:unjustify**.

```
(define-predicate is-uncle-of (uncle niece-or-nephew) (ltms:ltms-predicate-model))
(define-predicate is-nephew-of (nephew uncle) (ltms:ltms-predicate-model))

(defrule notice-uncles (:forward)
  if [is-uncle-of ?uncle ?nephew]
  then [and (format t "~&I note that ~A is the uncle of ~A." ?uncle ?nephew)
          [is-nephew-of ?nephew ?uncle]])
```

First we'll **joshua:tell** an avuncular fact, **joshua:untell** it, and then re-**joshua:tell** it. After the first **joshua:tell** the fact fires the forward rule. After the second **joshua:tell** the forward rule fires again, since **joshua:tell** sees the predication as **T**.

```
(setq canonicalized-uncle-fact (tell [is-uncle-of Judah Manasseh]))
I note that JUDAH is the uncle of MANASSEH.
[IS-UNCLE-OF JUDAH MANASSEH]
T

Show Joshua Database
True things
  [IS-UNCLE-OF JUDAH MANASSEH]
  [IS-NEPHEW-OF MANASSEH JUDAH]
False things
  None

(untell canonicalized-uncle-fact)
```

```
Show Joshua Database
True things
  None
False things
  None

(tell [is-uncle-of Judah Manasseh]) ; this fires the rule again!
I note that JUDAH is the uncle of MANASSEH.
[IS-UNCLE-OF JUDAH MANASSEH]
T
```

Now we'll use a variation of this example.

We start with the fact we just entered in the database above and which fired the forward rule. Now we **joshua:unjustify** the fact and then **joshua:tell** it again.

After the **joshua:unjustify**, the fact changes its truth value from **joshua:*true*** to **joshua:*unknown***, *but remains in the database*. When we **joshua:tell** the fact once again, its truth value changes from **joshua:*unknown*** to **joshua:*true***, but **joshua:tell** already knows about the fact, and no forward rules fire. Note, however, that the TMS brings the `is-nephew-of` deduction back in. We can tell it does so without re-executing the rule, since the side-effect (the **format** message) in the rule-body did not recur.

```
Show Joshua Database
 True things
[IS-UNCLE-OF JUDAH MANASSEH]
[IS-NEPHEW-OF MANASSEH JUDAH]
 False things
None

  (unjustify [IS-UNCLE-OF JUDAH MANASSEH])
NIL

Show Joshua Database
 True things
None
 False things
None

(tell [is-uncle-of Judah Manasseh])
 ; tell knows this fact is old, and it doesn't rerun the forward rule
[IS-UNCLE-OF JUDAH MANASSEH]
NIL
```

```
Show Joshua Database
 True things
[IS-UNCLE-OF JUDAH MANASSEH]
[IS-NEPHEW-OF MANASSEH JUDAH]
 False things
None
```

In sum, **joshua:unjustify** and **joshua:untell** do similar things, but with significant differences. If you want to change your mind about believing a fact but reserve your right to return to that fact later, you probably want to use **joshua:unjustify**. If, on the other hand:

- You just did a scratch calculation and want to flush it now that you have the answer, or
- You want the storage back, or
- You don't intend to come back and raise the issue of re-running rules.

you probably want to use **joshua:untell**.

Related Functions:

joshua:tell
joshua:unjustify
"Clear Joshua Database Command"

See the section "Removing Predications From the Database" in *User's Guide to Basic Joshua*.

See the section "The Joshua Database Protocol", page 8.

See the section "Customizing the Data Index", page 81.

joshua:value-of *slot value* — *Joshua Predicate*

This predicate is part of the Joshua object facility. It is used to assert and query the value of attributes of Joshua objects.

Values of the attributes of Joshua objects are maintained in data-structures called slots. The first argument to this predication must be either a slot or a path-name describing a slot.See the section "Using Paths to Refer to the Structure of an Object", page 109.

ltms:value-of *slot value* — *Joshua Predicate*

This predicate is part of the Joshua object facility. It is used in the same manner as **joshua:value-of**, except it refers to slots whose values are truth-maintained. Slots are declared as truth-maintained at the time the class of objects is defined by **joshua:define-object-type**.

joshua:variant *object1 object2* — *Function*

Two predications that differ *only in the names* of the logic variables they contain are equivalent, and are said to be *variants* of each other.

The function **joshua:variant** checks whether two objects are *variants* of each other. If they are, it returns **t**, otherwise **nil**.

When **joshua:tell** has to add a predication to the database it uses **joshua:variant** to determine if the predication is already there.

object1 A predication

object2 Another predication

"Variant" means there is a *renaming* of variables that makes one variable look like the other. For example:

```
(define-predicate foo (object))

(variant [foo 1 ?x] [foo 1 ?y])
T                                ; you can rename ?x → ?y.

(variant [father ?x ?y] [father ?a ?b])
T
```

joshua:variant should not be confused with **joshua:unify**. The latter tries to see if two objects can be *made to be* the same. **joshua:variant** checks if they *are* the same. It doesn't ever *set* logic variables, but merely looks for a renaming. **joshua:variant** is based on the notion that it should not matter what the names of logic variables are, so long as the structures are the same. This is a much stronger condition than **joshua:unify**. Thus, every pair that satisfies **joshua:variant** also satisfies **joshua:unify**, but not vice versa.

```
 (variant [foo 1 ?x] [foo 1 bar])
NIL                              ; these unify, but are not variants
                                 ; variables cannot be renamed

 (variant [father ?x ?x] [father ?a ?b])
NIL
```

joshua:variant also works on other structures such as lists.

Examples:

```
 (variant 'a 'a)
T

 (variant '([foo baz] [foo bar]) '([foo baz] [foo bar]))
T
```

See the section "Variables and Scoping in Joshua" in *User's Guide to Basic Joshua*.

joshua:with-atomic-action &body *body* *Macro*

Sometimes it is useful to be able to suspend forward rule triggering until the execution of a block of code has completed. The code might contain a number of **joshua:tell**'s and **joshua:untell**'s intermixed in such a way that the changes to the database are not coherent until the entire block of code has finished executing.

If the code is contained inside a **joshua:with-atomic-action** form, then no forward rule will be triggered until all of the code has executed. Furthermore, the rules that will be triggered are those whose trigger patterns are satisfied at the time that the code completes. Even if there was an intermediate point in the execution when a rule's trigger pattern was satisfied the rule will only run if there is a valid set of matching assertions at the time *body* has finished executing.

For example:

```
(defrule Test-Atomicity (:forward)
 If [and [P ?x ?y]
         [Q ?y]]
 Then (Print 'Foobar))

(tell [P 1 2])

(with-atomic-action
 (tell [Q 2])
 (Untell [P 1 2]))
```

In this case the rule Test-Atomicity will never trigger, even though in the middle of executing the with-atomic-action form it had a valid triggering set consisting of

```
[P 1 2]
[Q 2]
```

In this specific case the code is simple enough that one could simply have placed the **joshua:untell** before the **joshua:tell**. However, often the situations which require this form of control over rule invocation are also the ones that are complex enough that reordering the code to gain the right effect is too complicated.

With-Atomic-Action provides a simple means for treating the entire dynamic extent of a block of code as a single transaction to which the rule triggering mechanims react.

joshua:with-predication-maker-destructured *arglist predication-maker* &body *body* *Macro*

arglist	An arglist suitable for **destructuring-bind**
predication-maker	A predication-maker s-expression

body A body of code to execute.

This Macro is analogous to **joshua:with-statement-destructured**, but operates on predication-makers, rather than predications. It destructures the "statement" part of the predication-maker into the variables in the arglist and then executes the body in this environment. For example,

```
(read) [Foobar 1 ?x]
```

yields

```
(JI::PREDICATION-MAKER
  '(FOOBAR 1 (JI::LOGIC-VARIABLE-MAKER |?X|)))
```

and

```
(with-predication-maker-destructured (a b)
  (JI::PREDICATION-MAKER
   '(FOOBAR 1 (JI::LOGIC-VARIABLE-MAKER |?X|)))
 (print a)
 (print b))
```

would print:

```
1
and
(JI::LOGIC-VARIABLE-MAKER |?X|)
```

joshua:with-statement-destructured *arglist predication* &body *body* — *Macro*

Provides access to the *arglist* of *predication*. Wrap this macro around a *body* of code within methods in which you want to refer to the arguments of a predication that are not already in instance variables. (This macro works outside of methods, too.)

arglist The argument list of the specified *predication*. This can be anything suitable for **destructuring-bind**.

predication A Joshua predication.

For example, inside a **joshua:say** method for the predication **foo**:

```
(define-predicate enough-already (number-of servings food))

(define-predicate-method (say enough-already)
                         (&optional (stream *standard-output*))
  (with-statement-destructured (how-many servings food) self
    (format stream "~% You've just had ~A ~A of ~A. Hadn't you better quit?"
            how-many servings food)))
```

```
(say [enough-already 5 platters pickled-pigs-feet])
You've just had 5 PLATTERS of PICKLED-PIGS-FEET. Hadn't you better quit?
NIL
```

Related Functions:

joshua:define-predicate

joshua:with-unbound-logic-variables *variable-list* &body *body* *Macro*

This macro provides a way to generate a set of logic variables for use in code. Each (Lisp) variable within the *variable-list* is bound within the scope of the macro to a distinct, non-unified logic variable within the *body* of the macro. In essence a Lisp variable in *variable-list* has as its Lisp value a logic variable, for the duration of *body*.

variable-list
Is a list of variables

body Is any lisp form

Example:

The predicate `presidential-candidate` is defined in the following example. The macro is used to temporarily set *anybody* to be a logic variable. Then two predications are compared to see if they unify with one another. Unification occurs in this case so the format statement prints the value of *anybody*.

```
(define-predicate presidential-candidate (somone))

(with-unbound-logic-variables (anybody)
    (with-unification
      (unify `[presidential-candidate ,anybody] [presidential-candidate Abe])
      (format t "~&The value of anybody is ~s." anybody))))
The value of anybody is ABE.
NIL
```

joshua:with-unification &body *body* *Macro*

Establishes the scope within which substitutions specified by the **joshua:unify** function take effect. This temporary unifying mechanism is useful within Lisp code in the body of Joshua rules, since it lets the programmer try out a variety of different matching options.

Whenever unification fails, **joshua:unify** goes to the end of the dynamically innermost **joshua:with-unification** and undoes all the bindings established so far.

Thus, **joshua:with-unification** establishes both of the following:

- The scope of unifications done in its body
- A place to be thrown to if a unification in its body fails

Examples:

```
(define-predicate candidate-word (a-word))
(define-predicate is-computer-jargon (some-word))
(defvar *computer-jargon* '(foo bar baz quux))


(defrule jargon-finder (:backward)
  IF (typecase ?candidate-word
       (unbound-logic-variable
        (loop for word in *computer-jargon*
               doing (with-unification
                        (unify ?candidate-word word)
                        (succeed))))
       (otherwise
        (member ?candidate-word *computer-jargon*)))
  THEN [is-computer-jargon  ?candidate-word])

(ask [is-computer-jargon ?x] #'print-query)
[IS-COMPUTER-JARGON FOO]
[IS-COMPUTER-JARGON BAR]
[IS-COMPUTER-JARGON BAZ]
[IS-COMPUTER-JARGON QUUX]
```

Related Function:

joshua:unify

See the section "Pattern Matching in Joshua: Unification" in *User's Guide to Basic Joshua*.

joshua:write-backward-rule-matcher *rule-trigger variables-in-trigger environment name-of-pred-to-match* — *Generic Function*

rule-trigger — The source representation of a backward rule trigger. See the section "The Source Representaton of Predications and Logic-variables".

variables-in-trigger — The names of the logic variables which occur in this pattern.

environment — The compiler environment. This is needed in case this generic function needs to use a code-walker or otherwise expand macros in a specific compiler environment.

name-of-pred-to-match — The name of the variable by which the matcher code should refer to the predication it is matching.

Return Values:

form A code fragment to perform the match. *bindings*
A set of bindings that the rule compiler should wrap around the matching code. *used-data-stack-p*
Whether this code uses the data stack.

This protocol function is used to generate the matcher code corresponding to the trigger pattern of a backward rule. For example in the rule:

```
(defrule foobar (:backward)
  If [bar ?y ?z]
  Then [foo ?x ?y])
```

This method will be called, with the following arguments:

```
(JI::PREDICATION-MAKER
  '(FOO (JI::LOGIC-VARIABLE-MAKER |?X|)
        (JI::LOGIC-VARIABLE-MAKER |?Y|)))
 (|?X| |?Z| |?Y|)
<the environment>
JI::.GOAL.
```

Notice that the first argument is not a predication [foo ?x ?y] but its source representation, see the section "The Source Representaton of Predications and Logic-variables".

The backward rule compiler turns the trigger pattern of the rule (i.e. its Then-Part) into a code fragment which tests whether the query being posed unifies with the rule's trigger pattern. The If-part of the rule is transformed into a nested series of **joshua:ask**'s which attempt to find matches to the patterns in the If-part that are consistent with the bindings produced by matching the trigger (and which are mutually consistent). The transformation of the If-part is controlled by the **joshua:expand-backward-rule-action** protocol function. This protocol function controls the generation of the matching code corresponding to the trigger.

The rule compiler combines the results of these two protocol functions into a single function which performs the trigger unification and the **joshua:ask**'s. Primarily it adds code to create bindings for the logic variables and to build the queries corresponding to each pattern in the If-part. The rule compiler attempts to make this function as efficient as possible by using the system stacks to hold most of the data.

The **joshua:write-backward-rule-matcher** function returns three values: The first is a code fragment (which must be a single form) which performs the unifications necessary.

For example, the default method for this protocol function returns the following code fragment:

```
(JI::UNIFY-PREDICATION JI::.GOAL. PRED-1382)
```

Which checks that the query (i.e. JI::.GOAL.) matches the trigger pattern PRED-1382. You might wonder what PRED-1382 is; that information is contained in the second return value:

However, the second value specifies a set of bindings that the rule compiler should wrap around the generate code:

```
((FORM-1383 `(FOO ,|?X| ,|?Y|))
 (PRED-1382 (make-predication FORM-1383 :STACK)))
```

Which builds PRED-1382 on the stack. The third return value is **joshua::t** indicating that this code will need to use the data stack.

In most cases, you will not use this method if it forces you to resort to such arcane devices.

For the above pattern, a different set of return values could have been:

```
(let ((statement (predication-statement .goal.)))
  (unify (pop statement) 'foo)
  (unify (pop statement) |?X|)
  (unify (pop statement) |?Y|))

NIL

NIL
```

Which takes advantage of the fact that FOO predications have a fixed number of arguments. Thus if the query's predicate is FOO (the first thing checked), there will be exactly two other arguments and we need not check for the goal being either too long or too short.

Notice that in the code generated the logic variables in the pattern are referred to by their name (i.e. as LISP variables).

joshua:write-forward-rule-full-matcher *rule-trigger predicate-variable-name environment* *Generic Function*

rule-trigger
: The source representation of a forward rule trigger. See the section "The Source Representaton of Predications and Logic-variables".

predicate-variable-name
: The name of the variable by which the matcher code should refer to the predication it is matching.

environment
: The compiler environment. This is needed in case this generic function needs to use a code-walker or otherwise expand macros in a specific compiler environment.

This protocol function is used to generate the unification code corresponding to a specific forward-rule trigger pattern. For example in the rule:

```
(defrule foobar (:forward)
  If [and [foo ?x ?y]
          [bar ?y ?z]]
  Then <body>)
```

This method will be called twice, with the following arguments for the first call:

```
(JI::PREDICATION-MAKER '(FOO (JI::LOGIC-VARIABLE-MAKER |?X|)
                              (JI::LOGIC-VARIABLE-MAKER |?Y|)))
JI::PREDICATION-TO-MATCH
<the environment>
```

Notice that the first argument is not a predication [foo ?x ?y] but its source representation, see the section "The Source Representaton of Predications and Logic-variables".

The rule compiler produces two matchers corresponding to each trigger-pattern: The first performs unification and is invoked when the data being asserted contains logic variables; the second is invoked when the data contains no logic variables. This second matcher can be considerably more efficient than the first. Most predications asserted in the Joshua data base do not contain logic-variables, so it is useful to check for this case and use the more efficient matcher when possible.

The return value of this generic function is a code fragment (in particular a single form) which performs the unifications necessary to check that the rule's trigger pattern matches the data. The default method for this protocol function returns the following value:

```
(JI::UNIFY-PREDICATION (JI::PREDICATION-MAKER
                         '(FOO
                            (JI::LOGIC-VARIABLE-MAKER
                              |?X|)
                            (JI::LOGIC-VARIABLE-MAKER
                              |?Y|)))
                       JI::PREDICATION-TO-MATCH)
```

The rule compiler assembles this code into the complete matcher function by adding code that correctly interfaces this unification code with the rest of the rete network code.

joshua:write-forward-rule-semi-matcher *rule-trigger predicate-variable-name environment* *Generic Function*

rule-trigger — The source representation of a forward rule trigger. See the section "The Source Representaton of Predications and Logic-variables".

predicate-variable-name — The name of the variable by which the matcher code should refer to the predication it is matching.

environment — The compiler environment. This is needed in case this generic function needs to use a code-walker or otherwise expand macros in a specific compiler environment.

This protocol function is used to generate the matcher code corresponding to a specific forward-rule trigger pattern. For example in the rule:

```
(defrule foobar (:forward)
  If [and [foo ?x ?y]
          [bar ?y ?z]]
  Then <body>)
```

This method will be called twice, with the following arguments for the first call:

```
(JI::PREDICATION-MAKER '(FOO (JI::LOGIC-VARIABLE-MAKER |?X|)
                              (JI::LOGIC-VARIABLE-MAKER |?Y|)))
JI::PREDICATION-TO-MATCH
<the environment>
```

Notice that the first argument is not a predication [foo ?x ?y] but its source representation, see the section "The Source Representaton of Predications and Logic-variables".

The rule compiler produces two matchers corresponding to each trigger-pattern: The first performs unification and is invoked when the data being asserted contains logic variables; the second is invoked when the data contains no logic variables. This second matcher can be considerably more efficient than the first. Most predications asserted in the Joshua data base do not contain logic-variables, so it is useful to check for this case and use the more efficient matcher when possible.

This protocol function is used to generate the more efficient matcher.

The return value of this generic function is a code fragment (in particular a single form) which performs the semi-match.

This generated code fragment must check that the rule's trigger pattern matches the data. It also is responsible for producing variable bindings. Semi matchers do not need to use logic variables and unification (this is one reason they can be more efficient). Instead, the rule matcher creates a Lisp

variable corresonding to each logic-variable in the pattern. The semi-matcher is responsible for assigning a value to each of these variables and for checking that the assignments are consistent.

For example, the default method for this protocol function returns the following code fragment:

```
(LET ((THING-1306 (CDR (PREDICATION-STATEMENT JI::PREDICATION-TO-MATCH))))
      (AND (CONSP THING-1306)
           (PROGN (SETQ |?X| (CAR THING-1306)) T)
           (LET ((THING-1307 (CDR THING-1306)))
                (AND (CONSP THING-1307)
                     (PROGN (SETQ |?Y| (CAR THING-1307)) T)
                     (NULL (CDR THING-1307))))))
```

Notice that this code fragment returns **joshua::t** if the mactch succeeds and **joshua::nil** otherwise.

Also notice that this code fragment never checked whether the predicate of the predication being matched is the same as the predicate of the rule trigger. This is because the default data indexer has already guaranteed this and therefore the match generator knows that it need not emit code to perform this check; see the generic function **joshua:positions-forward-rule-matcher-can-skip**, page 214.

The rule compiler assembles this code into a complete matcher function by adding code that correctly interfaces to the rest of the rete network code.

It is possible to write protocol methods for this function which extend the matcher's syntax (e.g. by performing inline procedural checks as part of the match) and lead to increased efficiency. A good starting place for this is the default method provided with Joshua.

Index

Advanced Features of Joshua Rules, 24
A More Advanced Version of Mixed-chaining in **joshua:expand-forward-rule-trigger** 96
Basic Capabilities of the Joshua Object Facility, 107
Choosing Joshua Metering Types, 77
Clause Justification Structures, 65
Clear Joshua Database Command, 145
Compiling the Action Part of a Forward Rule, 27
conflict-resolution, 35, 254
Continuation Argument, 125
Controlling Choices in the LTMS, 67
Controlling Data and Rule Indexing, 79
Controlling Question Invocation, 47
Controlling Rule Invocation, 35
Customizing the Data Index, 81
Customizing the Data Index Without Storing Predications, 85
Customizing the Expansion of a Backward Rule, 99
Customizing the Expansion of a Forward Rule, 93
Customizing the Joshua Protocol, 5
Customizing the Matchers Generated by the Rule Compiler, 102
Customizing the Rule Compiler, 92
Customizing the Rule Index, 88
Dictionary Entries, 121
Difference between **joshua:untell** and **joshua:unjustify**, 248
Disable Joshua Tracing Command, 167
Displaying the database contents, 228
Enable Joshua Tracing Command, 169
Equalities Between Slot Values, 116
Examples of Using **joshua:ask**, 128
Explain Predication Command, 181
Extracting Parts of the Continuation with Accessor Functions, 126
Finding Backward Question Triggers, 50
Finding Backward Rule Triggers, 43
Finding Forward Rule Triggers, 41
Forward Rule Triggers: the Rete Network, 27

Graph Forward Rule Triggers Command, 184
Initial Values of Slots, 113
Introduction to the Joshua Object Facility, 105
Invoking Methods Associated with the Object Associated with a Slot, 115
ji:model-cant-handle-query flavor, 210
ji:model-only-handles-positive-queries flavor, 211
joshua:*contradictory* variable, 145
joshua:*false* variable, 183
joshua:*true* variable, 240
joshua:*unknown* variable, 248
joshua:act-on-truth-value-change generic function, 121
joshua:add-action generic function, 121
joshua:add-backward-question-trigger generic function, 122
joshua:add-backward-rule-trigger generic function, 122
joshua:add-forward-rule-trigger generic function, 123
joshua:ask function, 123
joshua:ask-data generic function, 133
joshua:ask-data method of **joshua:default-ask-model**, 136
joshua:ask-data-and-questions-only-mixin flavor, 134
joshua:ask-data-and-rules-only-mixin flavor, 134
joshua:ask-database-predication function, 135
joshua:ask-data-only-mixin flavor, 136
joshua:ask-derivation function, 137
Joshua Ask Metering, 75
joshua:ask-query function, 139
joshua:ask-query-truth-value function, 139
joshua:ask-questions generic function, 140
joshua:ask-questions-only-mixin flavor, 141
joshua:ask-rules generic function, 142
joshua:ask-rules-and-questions-only-mixin flavor, 143
joshua:ask-rules-only-mixin flavor, 143
joshua:basic-tms-mixin flavor, 144
joshua:clear function, 144
joshua:copy-object-if-necessary function, 146
joshua:database-predication presentation type, 148
joshua:default-ask-model flavor, 148
joshua:default-predicate-model flavor, 149

joshua:default-protocol-implementation-model flavor, 149
joshua:default-rule-compilation-model flavor, 149
joshua:default-tell-model flavor, 149
joshua:define-object-type macro, 149
joshua:define-predicate macro, 151
joshua:define-predicate-method macro, 152
joshua:define-predicate-model macro, 153
joshua:defquestion macro, 153
joshua:defrule function, 158
joshua:delete-backward-question-trigger generic function, 164
joshua:delete-backward-rule-trigger generic function, 165
joshua:delete-forward-rule-trigger generic function, 165
joshua:different-objects function, 166
joshua:discrimination-net-clear function, 167
joshua:discrimination-net-data-mixin flavor, 167
joshua:discrimination-net-fetch function, 167
joshua:discrimination-net-insert function, 168
joshua:discrimination-net-uninsert function, 169
joshua:equated joshua predicate, 170
joshua:equated-mixin flavor, 171
joshua:expand-backward-rule-action joshua protocol method, 178
joshua:expand-forward-rule-trigger generic function, 171
joshua:explain function, 181
joshua:fetch function, 184
joshua:graph-discrimination-net function, 184
joshua:graph-query-results function, 185
joshua:graph-tms-support function, 187
joshua:insert function, 189
joshua:justify function, 190
joshua:known joshua predicate, 191
Joshua Language Dictionary, 121
joshua:locate-backward-question-trigger generic function, 194
joshua:locate-backward-rule-trigger generic function, 196
joshua:locate-forward-rule-trigger generic function, 199
joshua:logic-variable-maker-name function, 203
joshua:logic-variable-maker-p function, 203
joshua:logic-variable-name function, 202
joshua:make-object function, 204

joshua:make-predication function, 204
joshua:map-over-backward-question-triggers generic function, 207
joshua:map-over-backward-rule-triggers generic function, 208
joshua:map-over-database-predications macro, 204
joshua:map-over-forward-rule-triggers generic function, 208
joshua:map-over-object-hierarchy function, 209
joshua:map-over-slots-in-object-hierarchy function, 209
joshua:map-over-slots-of-object function, 209
Joshua Merge Metering, 76
Joshua Metering, 73
Joshua Metering Types, 73
joshua:negate-truth-value function, 212
joshua:nontrivial-tms-p generic function, 213
joshua:notice-truth-value-change function, 213
joshua:no-variables-in-data-mixin flavor, 212
joshua:object-type-of joshua predicate, 213
joshua:part-of joshua predicate, 214
joshua:part-of-mixin flavor, 214
joshua:positions-forward-rule-matcher-can-skip generic function, 214
joshua:predication flavor, 216
joshua:predication presentation type, 216
joshua:predication-maker-p function, 216
joshua:predication-maker-predicate function, 217
joshua:predication-maker-statement function, 217
joshua:predicationp function, 218
joshua:predication-predicate function, 219
joshua:predication-statement function, 219
joshua:predication-truth-value function, 219
joshua:prefetch-forward-rule-matches function, 220
joshua:print-query function, 220
joshua:print-query-results function, 221
joshua:provable joshua predicate, 223
joshua:remove-action generic function, 225
joshua:say function, 226
joshua:say method of **joshua:predication**, 227
joshua:say-query function, 227
Joshua's Default Database: the Discrimination Net, 16

joshua:slot-current-predication generic function, 231
joshua:slot-current-value generic function, 231
joshua:slot-my-object generic function, 231
joshua:slot-value-mixin flavor, 232
joshua:succeed function, 232
joshua:support function, 233
joshua:support method of **joshua:basic-tms-mixin**, 234
joshua:support method of **joshua:default-protocol-implementation-model**, 234
joshua:tell function, 234
Joshua Tell Metering, 73
joshua:tms-bits generic function, 236
joshua:tms-contradiction flavor, 236
joshua:tms-contradiction-contradictory-predication generic function, 237
joshua:tms-contradiction-hard-contradiction-flavor generic function, 237
joshua:tms-contradiction-justification generic function, 238
joshua:tms-contradiction-non-premises generic function, 238
joshua:tms-contradiction-premises generic function, 238
joshua:tms-contradiction-support generic function, 239
joshua:tms-hard-contradiction flavor, 239
joshua:truth-value presentation type, 240
joshua:type-of-mixin flavor, 240
joshua:undefine-predicate macro, 240
joshua:undefine-predicate-method function, 241
joshua:undefine-predicate-model function, 242
joshua:undefquestion function, 242
joshua:undefrule function, 242
joshua:unify function, 243
joshua:uninsert generic function, 247
joshua:unjustify generic function, 247
joshua:unjustify method of **ltms:ltms-mixin**, 248
joshua:untell generic function, 248
joshua:value-of joshua predicate, 252
joshua:variant function, 252
joshua:with-atomic-action, 35
joshua:with-atomic-action macro, 254
joshua:with-predication-maker-destructured macro, 254
joshua:with-statement-destructured macro, 255

joshua:with-unbound-logic-variables macro, 256
joshua:with-unification macro, 256
joshua:write-backward-rule-matcher generic function, 257
joshua:write-forward-rule-full-matcher generic function, 259
joshua:write-forward-rule-semi-matcher generic function, 261
ltms:equated joshua predicate, 171
ltms:ltms-mixin flavor, 203
ltms:ltms-predicate-model flavor, 204
ltms:object-type-of joshua predicate, 213
ltms:part-of joshua predicate, 214
ltms:value-of joshua predicate, 252
Nogoods in the LTMS, 66
Notifying the LTMS of Contradictions, 70
occur-check done by unifier, 243
Ordering Rule Execution, 35
Organization of the Default Discrimination Net, 17
Other Capabilities of Slots, 113
Other Options in Define-Object-Type, 117
Overview of Advanced Joshua Concepts, 1
Part-Whole Hierarchy in the Joshua Object Facility, 112
Predications as Instances, 7
Reset Joshua Tracing Command, 225
Set Valued and Single Valued Slots, 113
Show Joshua Database Command, 228
Show Joshua Predicates Command, 228
Show Joshua Rules Command, 229
Show Joshua Tracing Command, 230
Show Rule Definition Command, 231
Signalling a Condition When **joshua:ask-data** or **joshua:fetch** Can't Handle a Query, 12
Signalling Contradictions and Managing Backtracking, 57
Signalling Truth Value Changes, 63
Slots and Attached Actions, 114
Slots and Truth Maintenance, 114
Storing and Retrieving Knowledge in Joshua: the Virtual Database, 7
Streamlining Typical Continuation Requests with Convenience Functions, 127
The Backward Rule Compiler, 33
The Contract of a Joshua TMS Justification, 55
The Contract of **joshua:add-backward-question-trigger**, 48

The Contract of **joshua:delete-backward-question-trigger**, 48
The Contract of **joshua:locate-backward-question-trigger**, 49
The Contract of **joshua:map-over-backward-question-triggers**, 50
The Contract of the Generic Function **joshua:clear**, 14
The Contract of the Generic Function **joshua:expand-backward-rule-action**, 100, 179
The Contract of the Generic Function **joshua:expand-forward-rule-trigger** 93
The Contract of the Generic Function **joshua:insert**, 9
The Contract of the Generic Function **joshua:uninsert**, 14
The Contract of the Generic Functions **joshua:ask-data** and **joshua:fetch**, 10
The Contract of the Joshua TMS Protocol Functions, 54
The Contract of the Trigger Adding Functions, 38
The Contract of the Trigger Deleting Functions, 38
The Contract of the Trigger Locating Functions, 39
The Contract of the Trigger Mapping Functions, 41
The Default Implementation of the Protocol, 2
The Forward Rule Compiler, 27
The Funtions of a Truth Maintenance System, 53
The Joshua Database Protocol, 8
The Joshua LTMS, 65
The Joshua Object Facility, 105
The Joshua Protocol of Inference, 2
The Joshua Question Facilities, 47
The Joshua Question Indexing Protocol, 48
The Joshua Rule Compiler, 26
The Joshua Rule Facilities
, 23
The Joshua Rule Indexing Protocol, 36
The Predicates Used in the Joshua Object Facility, 118
The Truth Maintenance Protocol, 54
TMS Utility Routines, 56
Truth Maintenance Facilities, 53
Type Hierarchy in the Joshua Object Facility, 110
Types of Truth Maintenance Systems, 54
Using **:ignore** in **joshua:expand-forward-rule-trigger**, 98

Using **joshua:expand-forward-rule-trigger**, 95
Using Paths to Refer to the Structure of an Object, 109
Using TMS Conditions: a Balance Beam Example, 58
weeding out self-referential behavior, 166
What is a Virtual Database?, 7
What the Backward Rule-compiler Does to the Actions of a Rule, 99, 178