Table of Contents

List of Figures

1. Introduction

1.1. About the Joshua Language

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

Joshua is a very compact system, organized around a small number of core functions. Joshua's default structures provide a simple declarative core language with built-in facilities for application development. Programming with the defaults is very straightforward, allowing you to build effective applications quickly. This is due to several features:

- The syntax of Joshua is uniform, statement-oriented, and Lisp-like, so that you need not learn an entirely new language.
- The interface to the database (any database) is simple and uniform, consisting of the three functions, **joshua:ask**, **joshua:tell**, and **joshua:clear**.
- Special Zmacs facilities like bracket matching and special characters ease program development.

User's Guide to Basic Joshua, the first manual in the Joshua documentation set, 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. We present all these topics in the companion documentation volume: See the document *Joshua Reference Manual*.

Joshua dovetails with Genera and Symbolics Common Lisp in much the same way that Lisp and Flavors do. Joshua itself is implemented with Flavors. Joshua is closely integrated with Lisp, and Lisp code can be used within Joshua rules. All of Genera's program development facilities are available to Joshua, namely, the Zmacs editor, Dynamic Windows, formatted output, debugging support, and the User Interface Management System.

1.2. About the Joshua Documentation

Joshua is a powerful and sophisticated tool that can be used at many levels, by people with varying AI programming experience, ranging from the relatively inexperienced to the expert. While the Joshua documentation is designed to help users at any level get the most out of Joshua, it is not an introductory AI text; we assume you have at least a passing acquaintance with AI programming concepts and terms.

Since Joshua is very closely integrated into its Symbolics Common Lisp environment, we also assume that you are familiar with the Genera facilities, Symbolics Common Lisp, and the Zmacs editor. Extensive documentation on these areas is provided elsewhere in the Symbolics documentation set.

The Joshua documentation consists of two manuals,

User's Guide to Basic Joshua and *Joshua Reference Manual*, as well as online documentation.

This division reflects a different task orientation as well as a different stage of familiarity with AI programming.

1.2.1. The User's Guide to Basic Joshua

User's Guide to Basic Joshua gives you everything you need to know to start developing Joshua applications.

The goal of this manual is to let you develop a feel for Joshua, together with the ability to do all of the most common programming tasks using only Joshua's built-in facilities. For this reason, we will work with a subset of the Joshua commands, and with only those top-level Protocol functions that are directly usable for standard operations and that do not represent special-purpose functions or subcomponents of the protocol. The manual is designed to answer questions such as "How do I interact with the Joshua database?", "What kind of reasoning operations can I do with Joshua?", and "How does Joshua handle Truth Maintenance?"

User's Guide to Basic Joshua covers the following topics:

- Predications
- Rules and Inference
- Questions
- Unification
- Using Joshua Within Lisp Code
- Justification and Truth Maintenance

User's Guide to Basic Joshua is organized into a conceptual discussion, followed by the "Basic Joshua Dictionary". This is an alphabetized dictionary of reference entries for the basic subset of Joshua symbols that let you program Joshua's default structures. Each entry provides a complete description of a single Joshua function or command, its syntax, what it returns, examples of its use, and cross-references to related functions or commands.

1.2.2. The Joshua Reference Manual

Joshua Reference Manual, the companion volume to this, is a much deeper presentation of the concepts you need to understand in order to put Joshua to the fullest possible use. This manual is for experienced programmers who need to write tailored, optimized applications.

Topics presented earlier are revisited here in more depth, with cross-references to the earlier manual. The focus here is on all levels of the Joshua Protocol functions, so that you can see how each works. With this understanding you can make changes to any component for efficiency. *Modeling*, that is, tailoring any of the Joshua components to your application, is introduced in some detail, including examples of modeling data structures, rule storage, and writing your own TMS.

These are some of the topics covered in

Joshua Reference Manual:

- The Database Protocol
- Trigger Indexing Protocol
- The Rule Compiler
- The TMS
- Modeling

As in the earlier volume, the conceptual discussion is followed by the "Joshua Language Dictionary", an alphabetized dictionary of reference entries for *all* Joshua symbols, methods, and commands. Each entry provides a complete description of a single Joshua function or command, its syntax, what it returns, examples of its use, and cross-references to related functions or commands.

Among the major goals of the documentation is to give you fast access to the information you need, to minimize your reading, and to let you work directly with the information presented. Topics are clearly cross-referenced to point you from basic to advanced coverage, as well as to related topics; important information is summarized visually in tables or figures for quick reference; the majority of the examples in the manual show output, and they can be yanked from Document Examiner into the Lisp Listener so that you can experiment with them yourself.

2. Getting Started with Joshua

2.1. Setting up the Joshua Context and File Attributes

In order to get to a correct Joshua package and to inform the Lisp reader that you are expecting it to deal properly with Joshua syntax, you must set your working context to Joshua.

From the Lisp Listener, enter:

```
Set Lisp Context Joshua
```

This sets the current context to use Joshua syntax, and sets the current package to Joshua. For information on Lisp contexts: See the section "Set Lisp Context Command" in *Genera Handbook*.

To set your file attribute list, enter the following from a Zmacs buffer:

m-X Create Initial Joshua Attribute List

This creates an attribute list similar to the one below:

;;; -*- Mode: Joshua; Package: JOSHUA-USER; Syntax: Joshua; Vsp: 0 -*-
;;; Created 3/18/88 10:24:45 by Covo running on LADY-PEREGRINE at SCRC.

For information on file attribute lists: See the section "Buffer and File Attributes in Zmacs" in *Editing and Mail*. You are now ready to use Joshua.

3. Overview of Joshua

Since the core of Joshua is a rule-based inference language, its chief building blocks might already be familiar to you from other AI programming languages. We will briefly review these elements as we present the Joshua context and terminology.

One can think of Joshua as having five major components:

Predications are ways of expressing knowledge; they are often called *statements*, *facts*, or *assertions* in other languages, and we occasionally use these terms as well.

The *database* is a collection of predications and related information that the system remembers;

Rules are ways of expressing and remembering relationships among predications, as well as procedural knowledge;

The *Protocol of Inference* is the mechanism that integrates these components and performs the reasoning;

The *Truth Maintenance System* (TMS) keeps track of the reasoning that was used so that it can:

- Maintain explanations of Joshua's reasoning
- Maintain the logical consistency of the system

Basic programming in Joshua consists of supplying predications and rules and determining how to use this knowledge to deduce additional knowledge, or how to use it to answer questions. We cover these topics in the present manual.

Advanced programming in Joshua consists of *modeling*, that is, tailoring appropriate parts of the Joshua Protocol routines to your particular application; you might want to model for any number of reasons (to increase efficiency, incorporate specialized tools, access external databases, and the like). We cover these topics in the

Joshua Reference Manual, the companion volume to this.

We begin with a summary of the Joshua protocol.

3.1. Some Basic Joshua Protocol Functions

The Joshua Protocol of Inference consists of some 30 generic functions broken down into five functional groups:

- **Database Interface**: Manages addition/deletion of facts to the database
- **Truth Maintenance System (TMS) Protocol**: Manages deductive dependencies
- **User Interface Protocol**: Manages interaction with the user
- **Rule Indexing Protocol**: Manages the rule database
- **Rule Customization Protocol**: Manages the rule compiler

Because of the modular nature of the Joshua Protocol, most Protocol functions are broken into subcomponents, each of which performs a specific part of the function's contract. This modularity lets you fine-tune and localize changes to functionality at any level, letting them remain transparent to the end user.

In this manual we cover some basic top-level protocol functions in the Database Interface, User Interface, and TMS groups. The functions for Rule Indexing and Rule Customization are used only for modeling, and are not of immediate interest here. Similarly, we will not be concerned with the details of modularization in each of the five protocol groups, since this information is only relevant for modeling.

Figure 1 is a depopulated picture of the basic protocol functions we are covering here. For the complete protocol: See the section "A Figure of the Joshua Protocol of Inference", page 167.

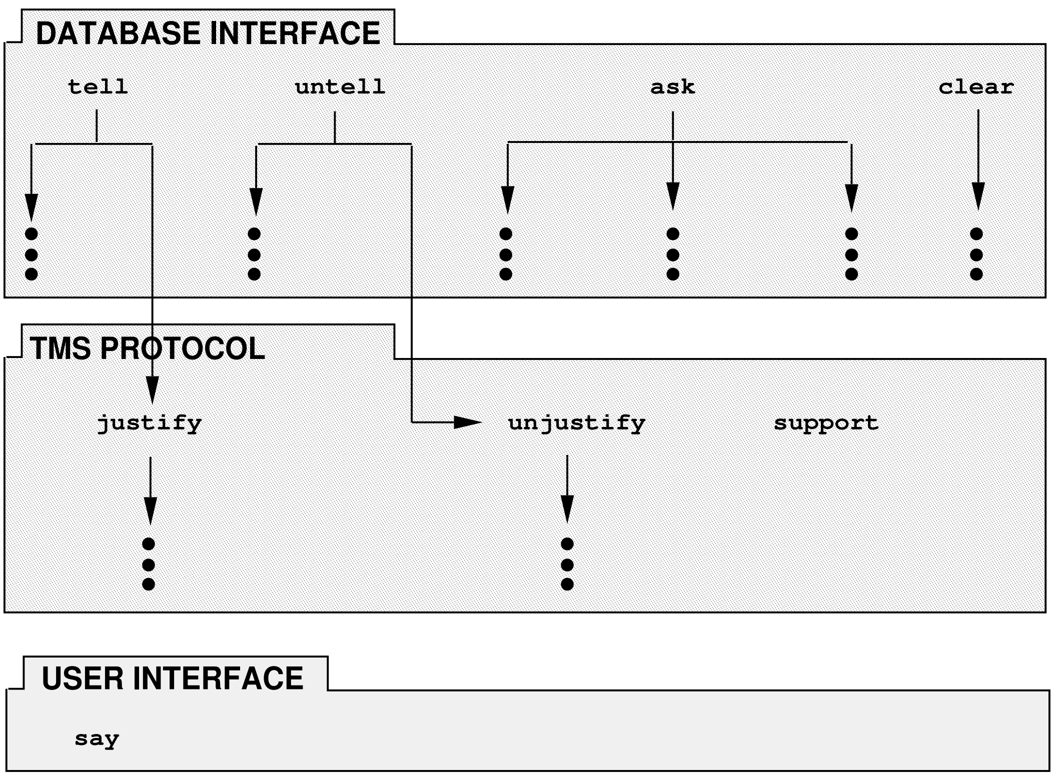


Figure 1. A Basic Subset of the Joshua Protocol

Here is a summary explanation of each of these functions.

Database Interface

- **joshua:tell**: Installs new information
  - **joshua:justify**: Gives the TMS a reason for believing the predication
- **joshua:ask**: Retrieves known or implied data
- **joshua:untell**: Removes a single fact that **joshua:tell** put into the database
  - **joshua:unjustify**: Manages the TMS issues related to removing the fact
- **joshua:clear**: Removes all facts from the database

TMS Protocol: Manages Deductive Dependencies

- **joshua:justify**: Installs a new TMS justification
- **joshua:unjustify**: Removes a TMS justification
- **joshua:support**: Finds the set of facts or assumptions that a statement depends on

User Interface

joshua:say: Prints out the meaning of predications in natural language, or something similar.

For a subset of Joshua symbols introduced in this manual: See the section "List of Basic Joshua Symbols", page 9.

3.2. List of Basic Joshua Symbols

Here is the list of functions and commands whose basic functionality we present in this manual.

joshua:ask
joshua:ask-database-predication
joshua:ask-derivation
joshua:ask-query
joshua:ask-query-truth-value
joshua:clear
"Clear Joshua Database Command"
joshua:*contradictory*
joshua:copy-object-if-necessary
joshua:define-predicate

joshua:defquestion
joshua:defrule

joshua:different-objects
"Disable Joshua Tracing Command"
"Enable Joshua Tracing Command"
"Explain Predication Command"
joshua:explain
joshua:*false*
joshua:graph-query-results
joshua:graph-tms-support
joshua:known
joshua:make-predication
joshua:map-over-database-predications
joshua:predication
joshua:predicationp
joshua:print-query
joshua:print-query-results
joshua:provable
"Reset Joshua Tracing Command"
"˜\\Say\\"
joshua:say
joshua:say-query
"Show Joshua Predicates Command"
"Show Joshua Rules Command"
"Show Joshua Tracing Command"
"Show Rule Definition Command"
joshua:succeed
joshua:tell
joshua:*true*
joshua:undefine-predicate
joshua:undefquestion
joshua:undefrule
joshua:unify
joshua:unjustify
joshua:*unknown*
joshua:untell
joshua:variant
joshua:with-statement-destructured
joshua:with-unbound-logic-variables
joshua:with-unification

To begin using Joshua, you need to work with *predications*, learning how to define, store, look up, and delete them. For these and related topics: See the section "Predications and Predicates", page 11.

4. Joshua Predications

AI and Joshua programming activities involve working with facts that represent knowledge. Programs build and store facts and reason with them, which leads to the building and storing (or removal) of other new facts to reason with.

Joshua facts are called *predications*. This chapter covers the essentials of building and storing predications. We also introduce related topics such as truth values associated with predications, and the use of logic variables to build *predication patterns*. Reasoning with predications is the province of rules and is covered in that context: See the section "Rules and Inference", page 41.

4.1. Predications and Predicates

Macro:	**joshua:define-predicate**
Function:	**joshua:undefine-predicate**
Command:	Show Joshua Predicates
Zmacs Command:	`m-X` Kill Definition

Predications are statements about the world. When predications are stored in a *database*, the knowledge they embody becomes available to the system, to be manipulated by rules.

Terminology note: Occasionally we use the terms *statement*, *assertion*, *fact*, and *belief* synonymously with *predication*. These are broadly equivalent terms that have wide currency in AI programming.

The protocol lets you manipulate predications in numerous ways. You can, for example:

- Insert them into a database
- Look them up in a database
- Infer them by rules
- Supply justifications that are remembered in the database

Bear in mind, also, that predications can be manipulated like other ordinary Lisp-world objects (you can store them in arrays, print them, **joshua::accept** them, and so on). In fact, predications are just flavor instances.

These are examples of predications:

```
[healthy John]
[has-pneumonia John]
[author-of (poems plays) Marlowe]
```

Predications consist of *predicates* such as `healthy`, `has-pneumonia` and `author-of` in the above example, together with their arguments, if any. A predication is always enclosed in brackets [ ].

Predicates thus are names of relationships, and they organize the knowledge in a predication and express relationships among its parts. A predicate is always the first item in a predication; it can take zero or more arguments, depending on how you define it.

You must define a predicate before using it in a predication. Defining the predicate sets up its format and specifies the kinds of information you expect to see after the predicate name (things like the arguments and, optionally, any customized ways of controlling the predicate's behavior); At runtime the system checks your predication's pattern against the format you set up in the predicate definition, and notifies you if there is a discrepancy.

Use the macro **joshua:define-predicate** to define new predicates. As an example, we'll define a predicate, `healthy`, to take a single argument, *object*. This predicate uses the default implementations for data storage and representation, so no further arguments are needed in the definition.

```
(define-predicate healthy (object))
```

Once a predicate has been defined, we can use it in any number of predications with different arguments. This makes it possible to organize related knowledge by grouping it together.

Note that you can use backquote with the bracket notation for predications. For example:

```
`[healthy ,(find-a-healthy-thing)]
```

Here are some examples of predications:

```
[healthy vegetables]
[healthy long-vacations]
[healthy "an apple a day"]
```

Since there is no one set way of thinking about a world, we can express a given piece of knowledge in a variety of different predications, depending on what aspects of it are important to our problem. We can, for instance, use several different predicates to express the fact that a person is healthy, or that a person is ill:

```
[healthy John]
[has-health John]
[has-pneumonia John]
[illness John pneumonia]
[pneumonia John]
```

Or we can relate health and illness conceptually as reflecting different states of being, and generalize them to a single condition by using a single predicate:

```
[has-condition John health]
[has-condition John pneumonia]
```

Or, if John is the only patient whose condition interests us, we can omit explicit mention of John's name from the statements altogether.

```
[has-condition health]
[has-condition pneumonia]
```

Thus, your choice of predicates reflects your analysis of the problem, the kinds of reasoning you expect your program to do, and the amount of knowledge you need to express explicitly or implicitly. Since predicates are your vocabulary for defining and exploring a knowledge domain, predicate selection is a major problem-solving challenge; choosing the correct set of predicates to model your knowledge can greatly simplify your programming task. Poor choices can lead to clumsy programs with too many rules.

You can "undefine" predicate definitions from either Zmacs or the Lisp Listener. (Undefining a predicate is a rare event that corresponds to a major reorganization of the program.) To remove a predicate definition while you are in Zmacs, place the cursor on the predicate definition and use the extended command m-X Kill Definition. In the Lisp Listener, use the function **joshua:undefine-predicate**.

For example:

```
(undefine-predicate 'healthy)
```

Note that while **joshua:undefine-predicate** removes the *predicate definition*, any *predications* built with this predicate remain in the world until you explicitly remove them. However, almost any attempt to use these predications will cause an error. (See the section "Removing Predications From the Database", page 17.)

To find out what predicates are defined in your current world, use the command Show Joshua Predicates. This prints the predicate names and their arguments. There are command options to let you tailor the display. Please consult the dictionary entry for this command.

Free-floating predications such as those we've just built are not usable for reasoning operations because the system does not keep track of them. To be on record, predications must be collected in a database that the system recognizes and remembers. Also, once predications are in a database, the system stores additional information about them, such as their truth value. The next section deals with these topics. See the section "Predications, Truth Values, and the Database", page 14.

Advanced Concepts Note:

In Joshua, predicates are an implementation-organizing as well as a knowledge-organizing tool. Since you can organize related facts in similar fashion by consistent predicate choice, it can also be convenient to implement them in the same way. So predicates, rather than their arguments, have specific protocol implementations associated with them. This means that in Joshua you can, if you wish, deal with each predicate independently, and give it its own unique way of interacting with the protocol; while defining a predicate you can specify such things as where predications using it are to be stored (this could be in any number of places, even on separate computer systems), or whether or not justifications for it are to be remembered. For some predicates you might want to specify algorithmic methods for answering questions about a problem, while for others you might request reasoning methods. In other words, you can *model* the behavior of each predicate, mixing and matching various built-in models, or adding your own.

The various models you might use are also independent of each other. You can, for example, define a TMS model (mostly) without affecting the storage model; in this way you need only change those portions of the protocol directly affected by your model. Similarly, your modeling choices require no modification of other knowledge-level structures, such as rules.

This flexibility is one of Joshua's major advantages. However, the extent to which you exercise it is strictly up to you, since it is possible to program effectively in Joshua using only the defaults.

All this is implemented using the generic functions and flavors mechanism of Symbolics Common Lisp.

We cover customized predicate models in *Joshua Reference Manual*.

4.2. Predications, Truth Values, and the Database

To make a predication available to Joshua, you **joshua:tell** it into a *database*. A database is an extensible collection of predications together with associated information about each predication. Predications are generated by the programmer or as a result of reasoning done by rules; they are entered into the database by the programmer (or by the system if they are inferred by rules). A database thus contains all the facts Joshua knows about at this point in time, and it changes dynamically with new facts being added or removed as a result of inference or computation.

Predications in the database differ from free-floating predications first because they are remembered by the system, and second, because they have additional information, such as truth values, associated with them. (See the section "Truth Values", page 20.)

As a Joshua user, your main interaction with the database takes place via two functions, **joshua:tell** and **joshua:ask**. **joshua:tell** stores statements into the database. (See the section "Entering and Displaying Predications in the Database", page 15.) **joshua:ask**, among its other functions, looks up knowledge in the database. (See the section "Querying the Database", page 23.)

Database operations include the following:

- Inserting predications into the database
- Displaying the database contents (text, or graphic format)
- Looking for predications in the database
- Removing predications from the database

This section covers the basics of the above operations. We also introduce the related topic of truth values that are associated with predications in the database.

Advanced Concepts Note:

The Joshua database protocol provides a default mechanism for data representation and storage. (See the section "The Joshua Database Protocol" in *Joshua Reference Manual*.) If you provide an alternate implementation of the relevant components of **joshua:tell** and **joshua:ask**, you can customize data indexing for your application. This will, for example, let you access knowledge in any existing database, or combinations of databases in various locations. The Joshua concept of a *virtual database* expresses this idea. (See the section "Customizing the Data Index" in *Joshua Reference Manual*.)

4.2.1. Entering and Displaying Predications in the Database

Functions: **joshua:tell**
joshua:untell
joshua:clear

Commands: `Show Joshua Database`
Clear Joshua Database

We use the Command Processor command `Show Joshua Database` to display the contents of the database. An empty database generates the following display:

```
Show Joshua Database
True things
  None
False things
  None
```

The display headings "True things" and "False things" are for grouping predications that the system knows to be true (those with a truth value of **joshua:*true***), and predications that the system knows to be false (those with a truth value of **joshua:*false***), respectively. (See the section "Truth Values", page 20.)

Let's define a new predicate:

```
(define-predicate has-eye-color (person color))
```

Now assume you or a rule application generate a predication such as:

```
[has-eye-color fred green]
```

This new predication is "free-floating", that is, it does not automatically become part of the database, unless you explicitly enter it there by giving it to **joshua:tell**.

joshua:tell takes the "free-floating" predication, and returns a *database predication*, that is, an object that it stored in the database and that can be accessed and reasoned with. It is important to note that this object, because it is stored with information about itself, is conceptually (and often physically) distinct from the "free-floating" predication you supplied as an argument to **joshua:tell**.

Whenever you need to do something with a stored predication (for example, to remove it from the database), you need to access the actual object that is stored. Later on we'll discuss ways of retrieving this object.

For clarity we use the terms *database predication*, whenever it is important to refer to the actual object that is stored in the database rather than to a predication in more general terms.

Using **joshua:tell** is very straightforward; you apply the **joshua:tell** function to the predication, as in this example:

```
(tell [has-eye-color Fred green])
[HAS-EYE-COLOR FRED GREEN]
T
```

joshua:tell returns the predication object it stored in the database. It also returns a boolean value indicating whether the predication is being inserted for the first time, as in the example above (**T**), or whether it already existed in the database (**nil**). This information is of interest primarily to the Truth Maintenance System, so we won't always reproduce it in the output from our examples; we mention it here only because you'll be seeing this every time you enter a **joshua:tell**.

Now display the database once more, and you will find that it includes the predication we have just entered with **joshua:tell**. `[has-eye-color fred green]` appears under "True things", indicating that Joshua knows the statement represented by the predication to be true.

```
Show Joshua Database
True things
  [HAS-EYE-COLOR FRED GREEN]
False things
  None
```

To indicate that a statement is false, prefix the predication representing that statement with the predicate **joshua::not**. For example, here we tell Joshua that we know Jane doesn't have green eyes.

```
(tell [not [has-eye-color Jane green]])
¬[HAS-EYE-COLOR JANE GREEN]
T
```

Here **joshua:tell** returns the database predication as usual. The Lisp printer prefixes the predication with the Not-sign logic symbol, (¬), denoting that the statement is false.

In the database display the predication appears under the heading "False things" without any prefix.

```
Show Joshua Database
True things
  [HAS-EYE-COLOR FRED GREEN]
False things
  [HAS-EYE-COLOR JANE GREEN]
```

Let's define a few more predications, add them to the database and then display it.

```
(define-predicate loves (lover beloved))

(tell [loves bonzo jane])
(tell [not [loves jane bonzo]])

  Show Joshua Database
True things
  [HAS-EYE-COLOR FRED GREEN]
  [LOVES BONZO JANE]
False things
  [LOVES JANE BONZO]
  [HAS-EYE-COLOR JANE GREEN]
```

The database display always shows the actual predications currently stored there. Click on any item in the display for use in Joshua commands that require database predications.

The `Show Joshua Database` command by default displays the entire contents of the database. This can become cumbersome with very large databases. Later on, we'll show how you can limit the database display by requesting only specific patterns or specific truth values. See the section "Predications and Logic Variables", page 26.

You can remove predications from the database as well as enter them.

See the section "Removing Predications From the Database", page 17.

Advanced Concepts Note:

The default database is implemented as a *discrimination net* stored in the value of the global variable **ji:*data-discrimination-net***. See the section "Joshua's Default Database: the Discrimination Net" in *Joshua Reference Manual*.

4.2.2. Removing Predications From the Database

There are several ways of removing predications from the database. Note that *predicate definitions* are not eliminated, but only the *predications* built with them.

The function **joshua:untell** removes a single predication from the database, and frees up some related storage. As its name implies, **joshua:untell** is the opposite of the function **joshua:tell**.

Another way to remove a predication from the database conceptually is to use the function **joshua:unjustify** if your application includes a TMS model. **joshua:unjustify** differs from **joshua:untell** in important respects. See the section "Retracting Predications with **joshua:unjustify**", page 84. The command 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.) You can specify predications matching a certain pattern, all predications, or none. You can also specify whether or not to remove predications that match the pattern specified but have the opposite truth value. (Using logic variables to build predication patterns is described elsewhere: See the section "Predications and Logic Variables", page 26.)

```
Command: Clear Joshua Database (predications, All, or None [default All]) All
Clear all predicates from the database? [default Yes]: Yes
```

To remove *all* predications from the database, use the function **joshua:clear**.

```
(clear)
Show Joshua Database

True things
  None
False things
  None
```

4.2.2.1. Removing a Database Predication with Untell

In the section "Entering and Displaying Predications in the Database", we mentioned that several Joshua operations require the actual predication object that is stored in the database. **joshua:untell** is one such operation. There are several ways of retrieving a predication object.

One method is to display the database contents and use the mouse to pick up the object you want to operate on. (Position the mouse cursor at the start of the line displaying the object, and click left on the mouse.) This picks up the object and positions it wherever your cursor is. The object so moved displays in capital italics (see example below).

For example, to remove the database predication `[has-eye-color Jane grey]`, display the database, enter (**joshua:untell**, move the cursor one space to the right, and click left on the target predication from the database display. Joshua inserts the predication at the cursor position (after the **joshua:untell**). Now enter a closing right parenthesis to end the command.

```
  Show Joshua Database (matching pattern [default All]) All
True things
  [HAS-EYE-COLOR JANE GREY]
  [HAS-EYE-COLOR FRED GREEN]
  [LOVES BONZO JANE]
  [FAVORITE-MEAL MONKEYS BANANAS]
False things
  [LOVES JANE BONZO]

  (untell [HAS-EYE-COLOR JANE GREY])  ;enter the predication by clicking left
                                      ;on the object in the database display
NIL


  Show Joshua Database
True things
  [HAS-EYE-COLOR FRED GREEN]
  [LOVES BONZO JANE]
  [FAVORITE-MEAL MONKEY BANANAS]
False things
  [LOVES JANE BONZO]
```

After the **joshua:untell** the database no longer contains the predication `[has-eye-color Jane grey]`.

Another way to get at the predication object is to specifically save the result of the **joshua:tell**. For example:

```
  (setq db-object-1 (tell [loves Narcissus Narcissus]))
[LOVES NARCISSUS NARCISSUS]

  Show Joshua Database (matching pattern [default All]) All
True things
  [HAS-EYE-COLOR FRED GREEN]
  [LOVES BONZO JANE]
  [LOVES NARCISSUS NARCISSUS]          ;new predication object is added
  [FAVORITE-MEAL MONKEYS BANANAS]
False things
  [LOVES JANE BONZO]

  (untell db-object-1)   ;db-object-1 is [loves Narcissus Narcissus]
NIL
```

```
    Show Joshua Database (matching pattern [default All]) All
  True things
    [HAS-EYE-COLOR FRED GREEN]
    [LOVES BONZO JANE]
    [FAVORITE-MEAL MONKEYS BANANAS]
  False things
    [LOVES JANE BONZO]
```

The predication `[loves Narcissus Narcissus]` is no longer in the database.

Other ways of retrieving a database predication involve database queries and using predication patterns. We'll get to these topics later.

See the section "Querying the Database", page 23.

See the section "Predications and Logic Variables", page 26.

4.2.3. Truth Values

A truth value is a value denoting what the system currently knows about the truth state of a database predication. A truth value becomes associated with a predication when Joshua adds the predication to the database. Predications commonly change their truth value as knowledge is updated.

A predication can be in any one of four possible truth states:

- **joshua:*true*** (appears under "True things" in the database display)
- **joshua:*false*** (appears under "False things" in the database display)
- **joshua:*unknown*** (does not appear in the database display)
- **joshua:*contradictory*** (a transient state; does not appear in the database display)

The truth value of a predication can be manipulated directly by the user. It can also be manipulated by the TMS (Truth Maintenance System). We cover the TMS in a separate section, touching on it here only as necessary. (See the section "Justification and Truth Maintenance", page 71.)

Joshua has a three-valued logic. A statement (predication) is **joshua:*true*** if its arguments are believed to satisfy the predicate, and **joshua:*false*** if it is known that they do not.

When making new predications, **joshua::not** is prefixed to a fact that is known to be **joshua:*false***, as in `[not [loves Jane Bonzo]]`.

If a fact is neither known to be **joshua:*true***, nor known to be **joshua:*false***, it is **joshua:*unknown***. In languages such as Prolog, if a fact cannot be proved to be true, it is assumed to be false. Joshua does *not* subscribe to this so-called "Closed World Assumption".

A predication is (or becomes) **joshua:*unknown*** when there is no valid reason that supports it. (For example, if a predication was **joshua:*true*** but the reason underlying this truth status is removed, the predication becomes

joshua:*unknown*.) The predication does remain physically in the database as an efficiency measure; however, since most reasoning operations look for a truth value of **joshua:*true*** or **joshua:*false***, a predication with a truth value of **joshua:*unknown*** is conceptually not seen. That is to say, from the point of view of the reasoning process, a predication that has a truth value of **joshua:*unknown*** is indistinguishable from one that is not in the database at all. When a predication's truth value changes from **joshua:*unknown*** to either **joshua:*true*** or **joshua:*false***, the predication is once more "visible" and used in the inferencing process.

The **joshua:*unknown*** truth state is used primarily by the TMS as it modifies the database for logical consistency.

As its name implies, the truth value of a predication is **joshua:*contradictory*** if there are reasons to believe that the predication is both true and false at the same time. This truth value also is primarily meaningful in conjunction with a TMS: if a TMS is present, it detects contradictory truth states and does not allow them to remain in the system. If no TMS is present, contradictions go undetected, and the most recent truth value is remembered. See the section "Revising Program Beliefs", page 77.

The truth value of a predication can change either because you explicitly modify it, or as a result of the monitoring activities of the TMS.

If you have not included a TMS in your predicate definition, you can change the truth value of a predication from **joshua:*true*** to **joshua:*false*** by reasserting the predication with **joshua:tell**. In this case, the system simply accepts the most recent fact you **joshua:tell** it, with no regard to logical consistency. Here, for example, are two **joshua:tell** statements.

```
(tell [loves Medea Jason])
[LOVES MEDEA JASON]
T

(tell [loves Medea her-children])
[LOVES MEDEA HER-CHILDREN]
T
```

Here's the database display containing these **joshua:*true*** predications.

```
  Show Joshua Database
True things
  [LOVES MEDEA HER-CHILDREN]
  [LOVES MEDEA JASON]
False things
  None
```

Say we now discover that Medea no longer loves her husband, Jason, and we want to modify the database to reflect this new fact. To change the truth value of `[loves Medea Jason]` from **joshua:*true*** to **joshua:*false***, re-enter the predication into the database, this time prefixing it with **joshua::not**:

```
(tell [not [loves Medea Jason]])
¬[LOVES MEDEA JASON]
NIL
```

A Truth Maintenance System would insist that Medea cannot both love and *not* love Jason simultaneously. Since we have no TMS in this example, Joshua accepts this new version of the predication without noting the contradiction. So the latest version entered (`[not [loves Medea Jason]]`), simply supersedes the old `version` (`[loves Medea Jason]`).

Note that the result of these two **joshua:tell** operations about Medea and Jason is the same database entry in each case. The Lisp printer prints the predication with the Not-sign symbol, (¬), the second time, to indicate that it changed its truth value. The second **joshua:tell** also returns the boolean **nil**, to indicate that the predication already existed in the database.

```
  Show Joshua Database
True things
  [LOVES MEDEA HER-CHILDREN]
False things
  [LOVES MEDEA JASON]
```

To change a predication's truth value from **joshua:*false*** to **joshua:*true***, use **joshua:tell** to reassert the predication into the database, without the **joshua::not** prefix.

Modifying truth values without using a TMS can introduce inconsistencies into your database. Since the system does not follow up the logical consequences of a change in truth value, any such inconsistencies remain undetected.

Suppose, for example, that the second predication in our database, `[loves Medea her-children]`, was deduced by some forward chaining rule, based on the belief `[loves Medea Jason]`. That is, the rule concluded that Medea loves her children because of the fact that she loves Jason.

Since we have now told the system that Medea does *not* love Jason, the conclusion that she loves her children no longer follows. (As we know, the mythical Medea went through some such thought process, killing her children to show her hatred of Jason.)

Without a TMS the system ignores these issues. Specifically, as we see from the last database display, it allows the fact `[loves Medea her-children]` to remain as a valid belief, even though it has lost its reason for being believed.

The TMS on the other hand follows up the logical consequences of a change in truth value and signals any resulting inconsistencies in the database. In this situation the TMS might have prevented infanticide, since asserting `[not [loves Medea Jason]]` would immediately cause a warning that Medea's love for her children is no longer a true fact. The TMS would then request you to correct the contradiction by removing one of the inconsistent statements.

As we can see, due to the TMS efforts to maintain a logically consistent database, adding a new predication can result in changed truth values for several existing predications. See the section "Revising Program Beliefs", page 77.

The Joshua predicates **joshua:known** and **joshua:provable** help us determine what we know. **joshua:provable** tells us whether a fact is currently known to have a particular truth value. **joshua:known** tells us if we have knowledge of a fact (regardless of whether the fact is **joshua:*true*** or **joshua:*false***). Both these predicates are more useful in their negative form (`[not [provable [...]]]`, and `[not [known [...]]]`). For some examples of their use, please consult the dictionary entries.

Predications combined with the standard predicate calculus connectives **and**, **or**, and **not** can be **joshua:*true*** or **joshua:*false***, according to the rules in the truth tables listing the truth value for each possible combination of argument truth values. See the section "Predications and Logical Connectives", page 31.

4.2.4. Querying the Database

Functions: **joshua:ask**
joshua:graph-query-results
joshua:print-query
joshua:print-query-results
joshua:say-query

Here are some predicate definitions and **joshua:tell** statements that create a simple database.

```
(define-predicate has-condition (object condition when))
(define-predicate adult (person))

(clear)

(tell [has-condition Fred measles today])
(tell [adult John])
(tell [adult Fred])
```

We need a way to query this database so that we can find out what Joshua knows. The generic function **joshua:ask** does this. The main purpose of **joshua:ask** is to find solutions to queries such as "Is John an adult?" "Who are the adults we know about?" "Do any adults currently have diseases?" It does this by looking in the database and by invoking backward rules (and questions) to derive answers by reasoning.

Because **joshua:ask** is your main interface to the knowledge and rule structures, many of its typical uses depend on concepts we have not yet covered; so, for the moment, we introduce **joshua:ask** in its most basic form. Further examples of its use appear throughout the rest of the manual.

See the section "Predications and Logic Variables", page 26. See the section "How Backward Rules Work ", page 46. See the section "Asking the User Questions", page 55. See the section "Justification and Truth Maintenance", page 71.

joshua:ask works as follows: you give it a *query pattern*, that is, some pattern like "John is an adult" that you want to validate. The query pattern is a predication that may, or may not, contain logic variables. (Logic variables are covered elsewhere: See the section "Predications and Logic Variables", page 26.)

joshua:ask looks for answers to the query, collecting information about its search process as it goes along.

As soon as it finds an answer, **joshua:ask** passes a list containing the answer, together with the related information about how the answer was derived, to a function called a *continuation*. The continuation can do what it likes with the answer.

The information received from **joshua:ask** resides in the single continuation argument called *backward-support*. *backward-support* is a list whose elements are: the answer to the query, the truth value of the query, and the derivation for the answer - that is, what line of reasoning was followed to determine the answer. Note that although the support is only one element of this list, we refer to the entire list as backward support. Also note that this list is usually stack-consed.

There are two ways of dealing with the backward support information. One way is to use Joshua's *accessor functions* in the **joshua:ask** continuation. These functions break up the backward support into separate elements and let you interpret these elements as you wish. We discuss accessor functions in the dictionary entry for **joshua:ask**.

An easier way of processing answers to queries is to use Joshua's *convenience functions* in the **joshua:ask** continuation. These functions extract parts of the backward support and interpret it for you. Generally we'll be using the convenience functions throughout this first manual. Here is a list of these functions. The first two deal with the answer to the query, and the other two interpret the reasons for the answer.

joshua:print-query Extracts and prints the answer to the query.

joshua:say-query Extracts the query and displays it in formatted form. See the section "Formatting Predications: the SAY Method", page 35.

joshua:print-query-results
Extracts and prints the support for the answer. The support varies depending on whether the answer was found in the database or derived from rules.

joshua:graph-query-results
Draws a graph of the query support, labeling rules and questions.

An additional convenience function, **joshua:map-over-database-predications** (instead of backward rules) is useful when you want to find answers only in the database and do some operation with those predications that you find. We mention this function here for completeness; for an example of its use: See the section "Predications and Logic Variables", page 26.

When Joshua is asked a question such as, `(ask [has-condition Fred measles today] ...)`, it tries to derive the answer from the database before looking for backward rules and questions to run. Here we discuss the database lookup.

When trying to answer a query, **joshua:ask** (conceptually) goes through the whole database, trying to match the predication pattern in the query against all predications in the database. We refer to pattern matching in Joshua as *unification*. A query pattern *unifies* with a predication either because the two patterns are *equivalent* (that is, they look the same and both predicate and arguments match exactly), as in

```
Query pattern:            (ask [has-condition Fred measles today] ...)
Database predication:          [has-condition Fred measles today]
```

or because appropriate values can be substituted for any variables in the two patterns to make them equivalent. See the section "Pattern Matching in Joshua: Unification", page 61.

When a query pattern succeeds in unifying with a predication, the query pattern becomes temporarily *instantiated* as that predication. That is, the logic variables in the query pattern are temporarily bound to the corresponding values in the database predication. At this point Joshua calls the **joshua:ask** continuation to process this answer. (The continuation is called once for each time the query can be satisfied.)

Here's the query, using the convenience function **joshua:print-query** in the continuation. Since the query can be answered (by finding a matching predication in the database), the continuation is called to process the answer. **joshua:print-query** displays the query with its variables instantiated, ignoring all the rest of the backward support information.

```
(ask [has-condition Fred measles today] #'print-query)
[HAS-CONDITION FRED MEASLES TODAY]
```

For illustration, let's use the continuation function **joshua:print-query-results** to show how Joshua derived the answer to our query.

```
(ask [has-condition Fred measles today] #'print-query-results)
[HAS-CONDITION FRED MEASLES TODAY] succeeded
  [HAS-CONDITION FRED MEASLES TODAY] was true in the database
```

As expected, Joshua tells us that the answer was found in the database, and returns the support for the satisfied query, that is, the actual database object that matched the query. When the answer is derived from rules, **joshua:print-query-results** traces the chain of reasoning that **joshua:ask** followed.

What would a graph of the support look like? Figure 2 shows the graph drawn by **joshua:graph-query-results** for the above query.

The graph tells us that the predication was in the database, and displays the database predication inside a rectangle. Rectangles in support graphs denote queries that were satisfied by the database (rather than by rules, or questions).

```
➪ (ask [has-condition Fred measles today] #'graph-query-results)
                 Database
[HAS-CONDITION FRED MEASLES TODAY]
```

Figure 2. Simple example using function graph-query-results

If you **joshua:ask** about a predication that is not in the database, and that cannot be derived from rules or questions, the system does not call the continuation and nothing is returned.

```
(ask [adult mary] #'print-query)
```

You can trace **joshua:ask** and **joshua:tell** operations in Joshua. See the section "Tracing Predications", page 37.

Asking questions about specific facts is of limited usefulness. It would be preferable to be able to ask the system more general questions, such as asking it to identify all the adults in our database. In other words, we need to **joshua:ask** the system questions using variables for which it will try to find the appropriate values. Joshua provides *logic variables* for this purpose. See the section "Predications and Logic Variables", page 26.

4.3. Predications and Logic Variables

A logic variable is a special type of object used by Joshua and recognized by Symbolics Common Lisp. Building predications with logic variables instead of constant values lets you create patterns; patterns let you move from the particular object to the level of generalized queries or statements.

If we are looking for a listing of all the adults the system knows about, we could **joshua:ask** Joshua about every single person we can think of:

```
(ask [adult Fred] #'print-query)
(ask [adult John] #'print-query)
.
.
.
```

Or (preferably), we could **joshua:ask** the question in a more general form such as, "Name everyone who is an adult". For example:

```
(ask [adult ?x] #'print-query)
```

`[adult ?x]` is a *query pattern* containing a predicate followed by a *logic variable*, `?x`. We don't know who `?x` might stand for, but Joshua can give us this information. A logic variable stands for something (in this case for a person); Joshua determines what the something is, depending on the query.

Joshua recognizes both the question-mark (?) and the equivalence symbol (≡) (typed `symbol-'`), by itself or followed by a symbol, as a logic variable:

```
[adult ?]
[adult ≡]
[adult ?person]
[adult ≡person]
[adult ?x]
[adult ≡x]
```

Regardless of whether you used logic variables or constants to enter your predication in the database, you can use logic variable arguments in an **joshua:ask** query. For example:

```
(define-predicate healthy (object))
(tell [healthy John])

(ask [healthy ?person] #'print-query)
[HEALTHY JOHN]
```

Choosing descriptive names for logic variables within query patterns is useful for documenting the predicate's arguments and their order.

Here are some **joshua:tell** statements about adults:

```
(define-predicate adult (person))
(tell [adult Fred])
(tell [adult John])
(tell [adult Rose])
```

If we **joshua:ask** the system to find all the adults in the `database (ask [adult ?person] ...)`, it does so by first matching the predicate `adult`, and then successively substituting the value of the predicate's argument (here a name), for the logic variable **?**`person` in the query. So queries with logic variables are satisfied by finding correct matches for the variables.

Since we want **joshua:ask** to look for solutions only in the database, we should prevent it from looking at backward rules or questions as well. We do this by using **:do-backward-rules nil**. For example:

```
(ask [adult ?person] #'print-query :do-backward-rules nil)
[ADULT ROSE]
[ADULT JOHN]
[ADULT FRED]
```

As we can see, the continuation is invoked three times, once for each time the query was satisfied. Let's walk through this example.

At the outset of the query, the logic variable **?**`person` in the query pattern `[adult ?person]` does not stand for any particular value; that is, it is as yet *uninstantiated*. An uninstantiated variable can match any object. So when Joshua searches through the database with an uninstantiated variable as an argument in the query, this variable is allowed to match any argument in the same position in the database predication, provided, of course, the predicates match.

The logic variable ?person in our example can match any first argument in a predication beginning with the predicate adult. When Joshua finds the first predication whose predicate is adult (here [adult Rose]), the argument, Rose, is temporarily substituted for the logic variable ?person in the query, so that ?person stands for Rose (becomes *instantiated* as Rose). In this way the query pattern matches the database predication. Joshua calls the continuation to print this answer as requested.

Once the continuation has finished executing, Joshua discards its temporary value assignments, resets logic variable ?person to be once more uninstantiated, and continues the search through the database to find the next match for the query, that is, another name that ?person could stand for. Joshua calls the continuation each time ?person is successfully instantiated. This process repeats until no more matching predications are found in the database.

Repeating the query and asking why it succeeded confirms that the query pattern was successfully matched against facts in the database.

```
(ask [adult ?person] #'print-query-results :do-backward-rules nil)
[ADULT ROSE] succeeded
  [ADULT ROSE] was true in the database
[ADULT JOHN] succeeded
  [ADULT JOHN] was true in the database
[ADULT FRED] succeeded
  [ADULT FRED] was true in the database
```

This is a somewhat simplified explanation of the pattern matching process. For more detail: See the section "Pattern Matching in Joshua: Unification", page 61.

Logic variables can substitute for any argument in the argument list of a query pattern and can be combined with other, non-variable arguments such as constants.

Let's add some statements about the health of people in our database, and ask various questions. For instance, to determine the current health status of the persons in our sample, the query can combine logic variables with a constant that specifies "today" as the time argument:

```
(tell [has-condition John measles last-year])
(tell [has-condition Rose pneumonia today])
(tell [has-condition Fred measles today])
(tell [has-condition Skip tendonitis today])
(tell [has-condition Fred cold last-week])
(tell [has-condition Marina sunburn today])
(tell [has-condition Fred infection last-month])
(tell [has-condition Dagwood hunger ?always]) ;note logic variable in database
                                              ;predication
```

```
(ask [has-condition ?person ?condition today] #'print-query :do-backward-rules nil)
[HAS-CONDITION MARINA SUNBURN TODAY]
[HAS-CONDITION SKIP TENDONITIS TODAY]
[HAS-CONDITION ROSE PNEUMONIA TODAY]
[HAS-CONDITION DAGWOOD HUNGER TODAY] ;variable in database is instantiated to
                                     ;constant in query
[HAS-CONDITION FRED MEASLES TODAY]
```

While matching `has-condition` predications, Joshua instantiates **?**`person` to the first argument in the predication and **?**`condition` to the second argument. Since we specified no logic variable for the third argument in the query pattern, Joshua matches it exactly. (Notice that since one database predication, namely `[has-condition Dagwood hunger ?always]` contains a logic variable, this variable became temporarily instantiated to the constant `today` in the query pattern.)

Here are some other examples.

```
(ask [has-condition Fred ?condition ?when] #'print-query :do-backward-rules nil)
[HAS-CONDITION FRED INFECTION LAST-MONTH]
[HAS-CONDITION FRED COLD LAST-WEEK]
[HAS-CONDITION FRED HEALTH YESTERDAY]
[HAS-CONDITION FRED MEASLES TODAY]
```

Logic variables have lexical scope and can be referred to by any Lisp code embedded in the **joshua:ask** continuation. The variables have their instantiated values until the continuation for a given answer has finished executing.

We can modify the continuation of our previous query about current health status to display a formatted answer using the logic variables as instantiated for each match:

```
(ask [has-condition ?person ?condition today]
     #'(lambda (ignore) (format t "~% ~A is ill with ~A." ?person ?condition))
     :do-backward-rules nil)
 MARINA is ill with SUNBURN.
 SKIP is ill with TENDONITIS.
 ROSE is ill with PNEUMONIA.
 DAGWOOD is ill with HUNGER.
 FRED is ill with MEASLES.
```

Adding some simple Lisp produces output with a heading:

```
(progn (format t "~% Today's Patient Status:")
       (ask [has-condition ?person ?condition today]
            #'(lambda (&rest ignore)
                (format t "~% ~A is ill with ~A." ?person ?condition)))
            :do-backward-rules nil)
```

```
 Today's Patient Status:
 MARINA is ill with SUNBURN.
 SKIP is ill with TENDONITIS.
 ROSE is ill with PNEUMONIA.
 DAGWOOD is ill with HUNGER.
 FRED is ill with MEASLES.
NIL
```

Giving a predication pattern to the convenience function **joshua:map-over-database-predications** is a very useful way of doing some operation on database predications that match the pattern. For example, suppose you are no longer interested in storing facts about persons with measles, so you want to find and remove these facts all at once. Give the appropriate pattern to **joshua:map-over-database-predications**. This function uses **joshua:ask** to search only the database, extracts the matching database predication from the **joshua:ask** continuation and executes the specified operation on that predication. In this case, we request that it **joshua:untell** every matching predication it finds.

```
(map-over-database-predications [has-condition ?person measles ?when] #'untell)

(cp:execute-command "Show Joshua Database")
True things
  [HAS-CONDITION DAGWOOD HUNGER ?ALWAYS]  [HAS-CONDITION FRED INFECTION LAST-MONTH]
  [HAS-CONDITION MARINA SUNBURN TODAY]    [HAS-CONDITION FRED COLD LAST-WEEK]
  [HAS-CONDITION SKIP TENDONITIS TODAY]   [HAS-CONDITION ROSE PNEUMONIA TODAY]
False things
  None
```

You can, of course, also use the Clear Joshua Database command with a predication pattern to remove matching predications from the database. To illustrate, suppose Fred has left this group of people we were monitoring, and all information about him needs to be removed from the database.

```
  Clear Joshua Database (predications, All, or None [default All])
  [has-condition Fred ?x ?when]
Predications being removed:
  [HAS-CONDITION FRED INFECTION LAST-MONTH]
  [HAS-CONDITION FRED COLD LAST-WEEK]
Untell the above predications? [default Yes]: Yes
```

Logic variables are also useful for narrowing down database displays to match only specified patterns. The default version of the `Show Joshua Database` command displays every item in the database.

```
Show Joshua Database (matching pattern [default All)]
```

If instead of the default you give the `Show Joshua Database` command a specific pattern to look for, Joshua displays only predications matching that pattern. This lets you isolate those portions of the database that immediately interest you. Additionally, when you specify a pattern to match, you can limit the display to matching predications of only a single truth value, **joshua:*true*** or **joshua:*false***. The default is both. To select one value, answer "No" to the question "(opposite truth-value too?)"

For example, we want to look at only those **joshua:*true*** predications that relate to hunger and ignore the rest of the database.

```
    Show Joshua Database (matching pattern) [has-condition ?person hunger ?when]
              (opposite truth-value too? [default Yes]) No
  True things
    [HAS-CONDITION DAGWOOD HUNGER ?ALWAYS]
```

Predication patterns as well as predications can be combined to express related ideas in combination. The next concept in our consideration of predications is that of logical connectives. See the section "Predications and Logical Connectives", page 31.

4.4. Predications and Logical Connectives

Predicates: **joshua::and**
joshua::or
joshua::not

So far we have been considering predications in their simplest form: a single predication made up of a predicate and its arguments. Such constructs have allowed us to express ideas in isolation, for example:

```
[has-condition Don-Quixote delusion today]
```

Knowledge is often more usefully expressed in some logical combination. For example, we might want to relate two ideas: that Don Quixote was suffering from delusions *and* that he believed windmills to be giants. Such combinations are made possible with the built-in Joshua predicates **joshua::and**, **joshua::or**, and **joshua::not**. So, after defining the appropriate predicates, we can write a compound predication such as:

```
[and [has-condition Don-Quixote delusion today]
     [believes Don-Quixote windmills giants]]
```

Nesting predications inside each other is another way of building more complex predications out of single ones.

You can use logical connectives in **joshua:tell** and **joshua:ask** statements (they are particularly useful, though, in rules and questions).

Compound **joshua:tell** statements using **joshua::and** just save some of the labor of entering multiple predications into the database. Once entered, the predications don't remain yoked together, that is, each of the component predications is inserted *individually*. You cannot use **or** with **joshua:tell** statements in the current release (**joshua:tell**ing a disjunction is a statement of partial knowledge).

joshua:ask queries can define more complex goals by using logical connectives.

To write compound **joshua:tell** and **joshua:ask** forms, wrap the predications with the appropriate connective. For example:

```
(tell [and [has-condition Don-Quixote delusion today]
           [believes Don-Quixote windmills giants]])
[AND [HAS-CONDITION DON-QUIXOTE DELUSION TODAY]
     [BELIEVES DON-QUIXOTE WINDMILLS GIANTS]]

(ask [or [has-condition ?person measles now]
         [has-condition ?person pneumonia now]] #'print-query)
[HAS-CONDITION FRED MEASLES NOW]
[HAS-CONDITION ROSE PNEUMONIA NOW]
```

Note that when you **joshua:tell** compound forms, **joshua:tell** does not return a boolean value.

For the next example, we build part of a library database of authors, book titles, and author information. We'll embed several Joshua operations inside a Lisp function that clears the database, enters the compound **joshua:tell** predications, and displays the newly created database. To display the database within Lisp code, we use the function `cp:execute-command "Show Joshua Database"`.

First, the predicate definitions.

```
(define-predicate author-of (work author))
(define-predicate lived (object when))
(define-predicate profession-of (person profession))
```

Next we build up part of the library.

```
(defun library ()
  (clear)
  (tell [and [author-of Tempest Shakespeare]
             [lived Shakespeare 16th-century]
             [profession-of Shakespeare actor]
             [author-of "Art of Love" Ovid]
             [lived Ovid 1st-century-BC]
             [profession-of Ovid poet]
             [author-of poems Henry-VIII]
             [lived Henry-VIII 16th-century]
             [profession-of Henry-VIII king]
             [author-of "Art of Love" Capellanus]
             [lived Capellanus 12th-century]
             [profession-of Capellanus chaplain]
             [author-of "Art of Love" Fromm]
             [lived Fromm 20th-century]
             [profession-of Fromm sociologist]
             [author-of poems Ralegh]
             [lived Ralegh 16th-century]
             [profession-of Ralegh soldier]])
  (cp:execute-command "Show Joshua Database"))
LIBRARY

(library)
True things
  [PROFESSION-OF RALEGH SOLDIER]       [LIVED HENRY-VIII 16TH-CENTURY]
  [PROFESSION-OF FROMM SOCIOLOGIST]    [LIVED OVID 1ST-CENTURY-BC]
  [PROFESSION-OF CAPELLANUS CHAPLAIN]  [LIVED SHAKESPEARE 16TH-CENTURY]
  [PROFESSION-OF HENRY-VIII KING]      [AUTHOR-OF POEMS RALEGH]
  [PROFESSION-OF OVID POET]            [AUTHOR-OF POEMS HENRY-VIII]
  [PROFESSION-OF SHAKESPEARE ACTOR]    [AUTHOR-OF "Art of Love" FROMM]
  [LIVED RALEGH 16TH-CENTURY]          [AUTHOR-OF "Art of Love" CAPELLANUS]
  [LIVED FROMM 20TH-CENTURY]           [AUTHOR-OF "Art of Love" OVID]
  [LIVED CAPELLANUS 12TH-CENTURY]      [AUTHOR-OF TEMPEST SHAKESPEARE]
False things
  None
```

Now we use a compound query to **joshua:ask** who wrote the book "Art of Love," when the author lived, and what the author's profession was; the continuation formats the answers into discursive English.

```
(ask [and [author-of "Art of Love" ?author]
          [lived ?author ?century]
          [profession-of ?author ?profession]]
   #'(lambda (ignore) (format t "~%~% ~A, a ~A ~A, wrote a version
                       of The Art of Love." ?author ?century ?profession)))
```

```
FROMM, a 20TH-CENTURY SOCIOLOGIST, wrote a version
                       of The Art of Love.

CAPELLANUS, a 12TH-CENTURY CHAPLAIN, wrote a version
                       of The Art of Love.

OVID, a 1ST-CENTURY-BC POET, wrote a version
                       of The Art of Love.
```

Logical connectives focus and refine queries and cut down the system's search time. In a compound **joshua:ask** using **joshua::and**, the logic variables that are common to all component predications must be instantiated to a common object for the query to succeed.

In our example, the logic variable **?**author is instantiated to a name matching the first query pattern, [author-of "Art of Love" ?author], namely Fromm.

?author remains instantiated to Fromm and the search for predications to match the rest of the queries above ([lived ?author ?century] and [profession-of ?author ?profession]), is narrowed down to only those predications matching [lived Fromm ?century] and [profession-of Fromm ?profession]. Only when all the predications in the compound query have been satisfied is the continuation called.

After the continuation executes, **?**author, ?century, and ?profession become uninstantiated in the first two predication patterns. The last predication pattern still has ?author instantiated. Joshua looks for the next solution to [profession-of Fromm ?profession]. It finds none, so it backs up and tries solutions to [lived Fromm ?century]. It finds none of these either, so it backs up and looks for solutions to the first query pattern, [author-of "Art of Love" ?author]. If we had **joshua:ask**ed the three questions separately,

```
(ask [author-of "Art of Love" ?author] ...)
(ask [lived ?author ?century] ...)
(ask [profession-of ?author ?profession] ...)
```

the system would have produced three separate lists containing a great deal of extraneous information, especially if we had a very large database: we would have gotten a list of authors of "The Art of Love"; a list of every author in the database and his century; and a list of every author in the database and his profession.

The logical connective **joshua::or** instantiates its logic variables separately for each predication in a query. The effect is the same as asking separate queries.

Logical connectives can, of course, be combined, as in this example. Here we look for all persons who are not children, and who are currently ill with either measles or mumps. (Assume the appropriate **joshua:tell** statements have been entered.)

```
(define-predicate child (person))
```

```
(ask [and [not [child ?person]]
          [or  [has-condition ?person measles today]
               [has-condition ?person mumps today]]]
     #'print-query :do-backward-rules nil)
[AND [NOT [CHILD HELEN]] [OR [HAS-CONDITION HELEN MEASLES TODAY]
[HAS-CONDITION HELEN MUMPS TODAY]]]
```

Now that we've seen how to build predications and predication patterns, it would be useful to display the meaning of predications in natural language. See the section "Formatting Predications: the SAY Method", page 35.

4.5. Formatting Predications: the SAY Method

Function: **joshua:say**

Format directive: **~\\SAY**

Macro: **joshua:define-predicate-method**

To print the meaning of a predication in natural language (or some other alternative, such as graphics), as opposed to the predicate calculus notation in which programs are written, you can use the **format** function, as we have been doing in some of our **joshua:ask** continuations; or we can embed formatting directives into a special method available for that purpose, the **joshua:say** method. **joshua:say** is actually a multi-purpose "hook" for using natural language, or graphics, or any other predicate-dependent approach you wish. Judicious use of **joshua:say** methods can make it easier to generate user interfaces quickly.

You define a **joshua:say** method by using the general method definition function, **joshua:define-predicate-method**. (This is the same function you would use to modify other parts of the Joshua protocol, and its advanced use is covered in the *Joshua Reference Manual*.)

The arguments to **joshua:define-predicate-method** are:

- The function-spec of the protocol method you are defining, for example, `(say foo)`.
- The argument list. In the case of **joshua:say** methods, this should almost always be `(&optional (stream *standard-output*))`.
- The body of the method.

The body of the method contains any Lisp code, presumably doing output to the stream mentioned in the argument list.

For example, suppose you want to write a **joshua:say** method for the predicate `[favorite-meal ...]`. This predicate takes two arguments, namely, `eater`, and `food`. You'll want to bind Lisp variables to the arguments of the predication so that you can use the arguments within the predicate method. To do this, wrap the macro **joshua:with-statement-destructured** around the body. The predicate arguments then become lexically available, and can be referred to by the Lisp forms. (If you have already defined the predicate with its instance variables destructured, you don't need to use **joshua:with-statement-destructured**. For more on methods of making predicate arguments lexically available: See the macro **joshua:define-predicate**, page 109.)

A **joshua:say** method for the predicate [favorite-meal ...] might look something like this:

```
(define-predicate favorite-meal (eater food))

(define-predicate-method (say favorite-meal) (&optional (stream *standard-output*))
  (with-statement-destructured (eater food) self
    (format stream
            "~& You can please ~A by giving them ~A to eat" eater food)))
```

Having defined this **joshua:say** method, you can use it in a variety of ways, as shown here. You can call the method directly, giving it an instantiated predication argument:

```
(say [favorite-meal monkeys bananas])
 You can please MONKEYS by giving them BANANAS to eat
NIL
```

Or you can use a **joshua::format** directive included in Joshua to call **joshua:say** instead of **joshua::prin1** or **joshua::princ**:

```
(format t "~% Is it true that: ~\\SAY\\?" [favorite-meal monkeys bananas])
 Is it true that:
 You can please MONKEYS by giving them BANANAS to eat?
NIL
```

Joshua provides the convenience function, **joshua:say-query** to display a satisfied query using either the default **joshua:say** method, or your own **joshua:say** method, if any. (The default **joshua:say** method simply prints out the answer to the query.)

Let's fatten up our database of foods and eaters before using the **joshua:say** method.

```
(defun eat-it ()
  (clear)
  (tell [and [favorite-meal bears honey]
             [favorite-meal mosquitoes people]
             [favorite-meal spiders flies]
             [favorite-meal Joshuas predications]
             [favorite-meal monkeys bananas]])
  (cp:execute-command "Show Joshua Database"))
```

```
(eat-it)
True things
  [FAVORITE-MEAL MONKEYS BANANAS]      [FAVORITE-MEAL MOSQUITOES PEOPLE]
  [FAVORITE-MEAL JOSHUAS PREDICATIONS]  [FAVORITE-MEAL BEARS HONEY]
  [FAVORITE-MEAL SPIDERS FLIES]
False things
  None

(ask [favorite-meal ?eater ?food] #'say-query :do-backward-rules nil)
 You can please MONKEYS by giving them BANANAS to eat
 You can please JOSHUAS by giving them PREDICATIONS to eat
 You can please SPIDERS by giving them FLIES to eat
 You can please MOSQUITOES by giving them PEOPLE to eat
 You can please BEARS by giving them HONEY to eat
```

Note: Technically our **joshua:say** method is correctly defined and it certainly works. But we purposely misphrased it, in order to make a point: the phrasing makes an implicit, new (and unwarranted) connection between the idea "favorite-meal" and the idea "to please by giving to eat". Much confusion in programming comes from such casual redefinition. A better phrasing would have been something like "The favorite meal of ... is ...".

4.6. Tracing Predications

Commands: Enable Joshua Tracing
Disable Joshua Tracing

Because database operations like **joshua:tell** and **joshua:ask** are so fundamental to the operation of Joshua programs, we often want to watch as things are stored into or retrieved from the database. You can trace the basic operations on predications by using the command:

```
Enable Joshua Tracing (type of tracing) Predications
```

This turns on the tracing of **joshua:ask** and **joshua:tell**. Each time your program calls **joshua:ask** or **joshua:tell**, the tracing facility prints a message saying which operation is being done and on what predication.

Example:

```
(defun tell-and-ask-foods ()
   (clear)
   (tell [and [favorite-meal bears honey]
              [favorite-meal mosquitoes people]
              [favorite-meal spiders flies]
              [favorite-meal Joshuas predications]
              [favorite-meal monkeys bananas]])
   (ask [favorite-meal ?eater ?food] #'say-query
        :do-backward-rules nil))
```

```
Command: Enable Joshua Tracing (Type of tracing) Predications

Predication tracing is on
   Tracing All predicates
   Traced events: Ask and Tell

Command: (tell-and-ask-foods)
▶ Telling predication [AND [FAVORITE-MEAL BEARS HONEY]
                           [FAVORITE-MEAL MOSQUITOES PEOPLE]
                           [FAVORITE-MEAL SPIDERS FLIES]
                           [FAVORITE-MEAL JOSHUAS PREDICATIONS]
                           [FAVORITE-MEAL MONKEYS BANANAS]]
▶ Telling predication [FAVORITE-MEAL BEARS HONEY]
▶ Telling predication [FAVORITE-MEAL MOSQUITOES PEOPLE]
▶ Telling predication [FAVORITE-MEAL SPIDERS FLIES]
▶ Telling predication [FAVORITE-MEAL JOSHUAS PREDICATIONS]
▶ Telling predication [FAVORITE-MEAL MONKEYS BANANAS]
▶ Asking predication [FAVORITE-MEAL =EATER =FOOD]
You can please MONKEYS by giving them BANANAS to eat
You can please JOSHUAS by giving them PREDICATIONS to eat
You can please SPIDERS by giving them FLIES to eat
You can please MOSQUITOES by giving them PEOPLE to eat
You can please BEARS by giving them HONEY to eat
```

In order to disable the tracing of predications use the command:

```
Disable Joshua Tracing (type of tracing) Predications
```

This turns off all tracing of predications.

You can get a greater degree of control over tracing by using the menu option to the Enable Joshua Tracing command:

```
Enable Joshua Tracing (type of tracing) Predications :Menu Yes
```

This shows a menu of all of the tracing options available for predications, letting you trace only predications matching a particular pattern, only predications using a particular predicate (or type of predicate), and to specify at which events you would like to see trace information.

In addition to the above commands you can set some tracing options by mousing right on a predication. This gives you a menu of the available options.

4.7. Miscellaneous Predication Facilities

Several predication facilities are available for use within Lisp code. We have already used one, namely, **joshua:with-statement-destructured**. Here are some others. We suggest you look up their dictionary entries for more detail.

joshua:different-objects If the arguments are **joshua::eql**, or if either argument is an uninstantiated logic variable, **joshua:different-objects** returns **nil**. Otherwise it returns **t**.

joshua:make-predication Constructs a predication (does not enter it into the database). The [ ] syntax is a reader macro that expands into this.

joshua:predication The base flavor for all predications in Joshua. This works well with functions like **joshua::typep** and **joshua::typecase**.

joshua:predicationp Checks whether an object is built on **joshua:predication**.

joshua:with-statement-destructured
Lets you bind Lisp variables to predication arguments.

This chapter has shown you how to generate information in Joshua by building predications, storing them in a database, and searching the database to answer queries. The next major concept is using Joshua to derive information by reasoning about predications with rules and questions. See the section "Rules and Inference", page 41.

5. Joshua Rules and Inference

Macro:	**joshua:defrule**
Function:	**joshua:undefrule**
Command:	Show Joshua Rules
Zmacs command	m-X Kill definition
Concepts:	Forward Chaining (**joshua:tell**) Backward Chaining (**joshua:ask**)

Often the information one looks for by searching the database is not there explicitly, but must be inferred by reasoning about the knowledge that is already there.

While predications define the relationships between objects and supply us with information about the domain, rules define the *reasoning* that is performed about predications, and control how we deduce knowledge from existing knowledge.

A rule is an independent piece of declarative and procedural information that determines how Joshua responds to a specified set of circumstances. Unlike conventional programming constructs, rules automatically execute in the proper order whenever the appropriate circumstances occur, rather than executing when a program's control structure reaches a specific portion of the code. In other words, rules can execute at any time, regardless of the order in which they are written.

The reasoning done by rules involves either *forward* or *backward* chaining. Joshua programs can use either or both of these rule types.

Forward chaining is *data-directed inference*, that is, reasoning from *known* facts to some conclusion. This form of reasoning says: "I now know fact X. What can I conclude from this?" For example, given the facts that birds can fly and that Tweety is a bird, a forward chaining rule can deduce the new fact that Tweety can fly. Thus, forward chaining is instrumental in adding to the database.

Forward chaining is activated by **joshua:tell**. That is, whenever you **joshua:tell** Joshua a new predication, the system looks for forward chaining rules that it can use (combined with knowledge already in the database) to draw conclusions from the new knowledge you gave it.

Backward chaining is *goal-directed inference*, that is, reasoning to satisfy some desired conclusion. This form of reasoning says: "I want to know fact X. How do I determine its validity?" For example, given the goal of determining whether Tweety can fly, a backward chaining rule would look for the facts it needs in order to support this goal, asking, for example, whether birds can fly and whether Tweety is a bird.

Backward chaining is activated by an **joshua:ask**. The *query pattern* becomes the *goal*, and the system then looks for the facts and rules and questions that might substantiate this goal. Backward chaining is thus useful in helping you determine the validity of a conclusion or goal.

Forward and backward rules look the same, except for the keyword **:forward** or **:backward** that indicates the rule's *control structure* (its inference method). If the rule doesn't use Lisp code, it should work equally well in either direction. Either method of reasoning accomplishes the same result; the choice of inference method depends on the problem being solved. Some problems are *much* more efficiently approached with one control structure than with the other.

This chapter summarizes basic information about forward and backward rules in Joshua.

Advanced Concepts Note:

The Joshua protocol has functions that determine how rules are stored, deleted, and looked up. See the section "The Joshua Rule Indexing Protocol" in *Joshua Reference Manual*. If you provide a consistent alternate implementation of these generic functions, you can customize rule management for your application. See the section "Customizing the Rule Index" in *Joshua Reference Manual*.

5.1. Defining Joshua Rules

A rule is defined with **joshua:defrule**. A rule has a *name*, a keyword argument specifying its *control structure* (whether it is a forward or a backward chaining rule), and a combination of patterns divided into an *if*-part and a *then*-part.

The control structure argument **:importance** lets you prioritize rules (the higher the value you give to **:importance**, the higher the priority). Higher priority rules run before lower priority rules.

Another control structure argument, **:documentation**, lets you add a documentation string explaining what the rule does. Use the Lisp function **documentation** to retrieve the rule's documentation string.

For example, this forward chaining rule describes some facts that let you deduce an identity for an unknown creature.

```
(defrule  dragon-id-kit (:forward :documentation "Identifies dragons")
  if [and [huge ?creature]
          [breathes ?creature fire]
          [or [guards ?creature gold]
              [guards ?creature maiden]]]
  then [dragon ?creature])

(documentation 'dragon-id-kit)
"Identifies dragons"
```

This backward chaining rule describes how to compute the grandfather relationship along paternal lines.

```
(defrule paternal-grandfather (:backward)
  if [and [father ?person ?dad]
          [father ?dad ?gramps]]
  then [grandfather ?person ?gramps])
```

Except for their control structure, both rules look similar.

A rule's *if-then* parts are its *logical* structure. The *if*-part of a rule logically describes conditions under which the rule is applicable. The *then*-part of a rule logically describes the rule's conclusions.

The *if*- and *then*- clauses in a rule can occur in any order. That is, both

```
if [...] then [...]
```

and

```
then [...] if [...]
```

are valid.

While the *form* of the *if-then* parts is identical for forward and backward rules, *procedurally* the parts differ for each rule type. This is because the logical structure maps into an *imperative* structure that reflects the inferencing method being used (forward or backward chaining). Forward and backward chaining rules differ in their mapping of logical to imperative parts.

A rule's imperative parts consist of a *trigger* part and an *action* part. The trigger part determines if a rule is applicable to a given situation. The action part determines what operations the rule performs when it executes. Forward and backward rules have different trigger and action parts, so we discuss each rule type separately.

The command Show Joshua Rules displays the currently defined rules. Various options let you tailor the display. Please consult the dictionary entry for this command.

5.1.1. How Forward Rules Work

In a forward rule, the *if*-part is the trigger, and the *then*-part is the action. See figure 3.

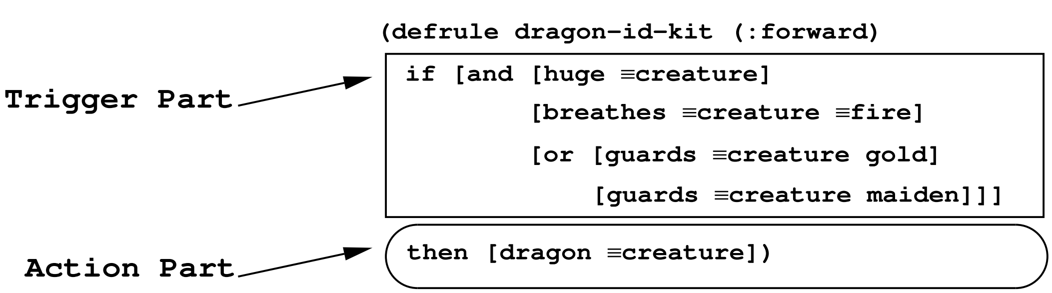


Figure 3. Forward Rule Trigger and Action Parts

In a forward rule the trigger is a single (possibly a compound) predication pattern stating conditions that must be satisfied for the rule to fire. The trigger can also contain Lisp code which is discussed below.

Data-directed inference is triggered by the addition of *new* facts into the database with **joshua:tell**. That is, new facts cause Joshua to look for forward rule triggers that can be satisfied by these facts.

What are new facts? A fact is new only when you **joshua:tell** it for the first time. A fact that you **joshua:tell** more than once is no longer new knowledge. Similarly, a fact that you **joshua:unjustify** and then **joshua:tell** again is not new knowledge, since it was never removed from the database. In contrast, a fact that you remove from the database (**joshua:untell**) and then **joshua:tell** again *does* represent new knowledge for Joshua.

There is no special order in which forward rule patterns are triggered. This is determined entirely by the order in which the new facts are entered.

To satisfy part of a rule's trigger, the pattern of that trigger must match that of the newly added database predication. This matching, called *unification* in Joshua, happens either because the two patterns are already equivalent, or because Joshua succeeded in finding substitutions for logic variables so that the two patterns become equivalent. To understand the basics of forward rule operation we can postpone looking at the details of unification. We cover this topic in the section"Pattern Matching in Joshua: Unification" along with the scoping rules that determine when two logic variables with the same name are the same and when they differ.

If a rule's trigger parts are joined by **and**, all conditions must be satisfied for the rule to trigger. If the trigger parts are joined by **or**, satisfying any of the conditions triggers the rule.

Note that backward rules are not automatically invoked while trying to satisfy a forward trigger. Since you can use Lisp code in forward triggers, your forward rules can explicitly call `(ask [foo ?x] ...)` in order to use a backward rule to satisfy `foo`.

When all of a forward rule's conditions (*if*-parts) are satisfied, the rule is fully triggered; it then *fires* and executes the action(s) in its action (*then*) parts. The action part can specify any action, including the new facts to be deduced directly from the current facts. If the action involves deducing a new fact, that fact is automatically added to the database (that is, a **joshua:tell** is implicit for forward rule deductions).

When the firing of forward chaining rules results in the addition of new facts to the database, this in turn can cause more forward rules to be triggered and fired, generating chains of conclusions until no more new facts can be generated.

Can you define a forward rule *after* adding a new fact that will trigger the rule into the database? The answer is that you can enter new rules and new facts in any order, because Joshua ensures that the right forward rules always get triggered.

Here's an example of forward chaining in Joshua. Rule `danger-sign` says that if a person is known to be a smoker and to have hypertension, then we can deduce that this person is in a high-risk category. We also want the system to notify us if it makes such a deduction so that we can take some appropriate action.

```
(define-predicate smoker (person))
(define-predicate has-condition (person condition when))
(define-predicate at-risk (person))
(defrule danger-sign (:forward)
  if [and [smoker ?person]
          [has-condition ?person hypertension today]]
  then [and [at-risk ?person]
            (format t "Suggest to ~S that smoking is dangerous to hypertensive persons"
             ?person)])
```

Our database already contains the predication `[smoker Ashley]`. When that predication was first added, it satisfied the first *if*-part of rule `danger-sign`, namely `[smoker ?person]`. Now we **joshua:tell** the system that Ashley is suffering from hypertension. This satisfies the rule's second *if*-part and triggers the rule. The rule fires and executes its *then*-part (the action part). This causes it to display a message and to add the newly deduced predication, `[at-risk Ashley]`, to the database.

```
(tell [has-condition Ashley hypertension today])
Suggest to Ashley that smoking is dangerous to hypertensive persons
[HAS-CONDITION ASHLEY HYPERTENSION TODAY]

  Show Joshua Database (matching pattern) [at-risk ?person]
True things
  [AT-RISK ASHLEY]
False things
  None
```

Assume we have another forward rule, `preventive-care`, as follows:

```
(define-predicate needs-checkup (person when))

(defrule preventive-care (:forward)
  if [at-risk ?person]
  then [needs-checkup ?person monthly])
```

The addition of the new fact, `[at-risk Ashley]`, to the database now satisfies the rule trigger of the above rule, causing it to fire in its turn, and to generate yet another new fact, namely, `[needs-checkup Ashley monthly]`.

```
  Show Joshua Database (matching pattern) [needs-checkup ?person ?frequency]
True things
  [NEEDS-CHECKUP ASHLEY MONTHLY]
False things
  None
```

This chain of inferences continues as long as there are forward rules that can be fully triggered by newly generated facts.

Forward chaining rules can contain Lisp code in both their *if-* (trigger) and *then-* (action) parts. Such Lisp code can refer to any logic variables that appear inside the body of the rule.

Our earlier rule, `danger-sign`, for example, uses Lisp in its action part. Lisp code in the *then*-part of a forward rule is just put into the rule body, to be run when the rule fires.

Here is an example using Lisp code in a forward rule trigger. We refer to such code as a *procedural node*. Procedural nodes can introduce new variables, have side-effects, call **joshua:succeed**, and so on.) Lisp code in the *if*-part of a forward rule returns non-**nil** if it wants the match to continue.

```
(defrule acceptable-price-rule (:forward)
  if [and [price-ceiling ?available-cash]
          [todays-price ?cost]
          (≤ ?cost ?available-cash)]
  then [acceptable-price ?cost])
```

You can watch forward rule execution by enabling Joshua tracing. See the section "Tracing Rules", page 50.

5.1.2. How Backward Rules Work

Since backward rules are goal-directed inference, it is the rule's *then*-part that specifies the desired goal. Thus, backward rule inferencing is triggered by the rule's *then*-part, while the *if*-part is the action part. See figure 4.

Note that the order of the *if-then* clauses does not matter. Those who like to place the *then*-part of backward rules first, so that the trigger always comes first, can safely do so.

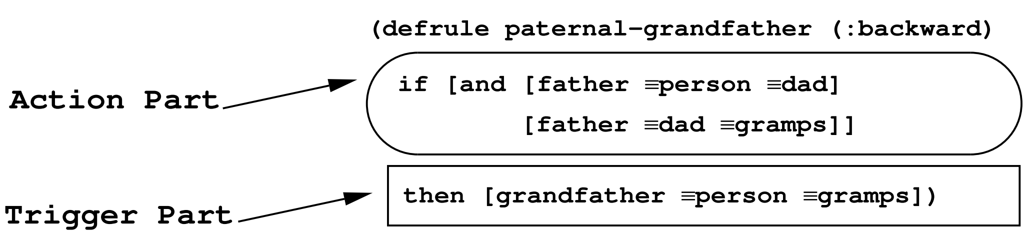


Figure 4. Backward Rule Trigger and Action Parts

Currently, backward rule inferencing is triggered by a single goal, that is, a backward trigger must be a single predication pattern. This restriction may be removed in the future.

A backward rule can contain Lisp code in its action (*if-*) part, but not in its trigger (*then-*) part. For example:

```
(defrule good-condition-rule (:backward)
  if (not (eql ?condition 'rusted))
  then [good-condition ?condition])
```

In the section "Querying the Database", we introduced **joshua:ask** for the limited purpose of looking up items in the database. Typically, the main purpose of **joshua:ask** is to find solutions through reasoning (finding backward rules and questions to run) as well as through database lookup.

To satisfy a query, **joshua:ask** always looks in the database first, trying to match the query pattern with a database predication. Next, **joshua:ask** by default searches for backward chaining rules whose trigger pattern (*then*-part) matches the query pattern. When such a match occurs, the backward rule is triggered. In Joshua, this pattern matching is called *unification*. For more on the mechanics of unification and the scoping of variables: See the section "Pattern Matching in Joshua: Unification", page 61.

Once a backward rule is triggered, it processes its *if*-parts (the action parts) from the top in the order given, searching for facts to validate the desired conclusion. (If the *if*-part is a compound predication joined by **and**, each component becomes a subgoal that is the subject of an implicit **joshua:ask**.)

The rule *succeeds* when all its *if*-parts (subgoals) have been satisfied. Satisfied means that there is a fact in the database that can serve as the bottom of the support structure for each subgoal.

joshua:ask collects information describing the solution process; when it finds a solution, **joshua:ask** calls its continuation, passing it the answer, along with the information collected about it. The continuation then executes on this result. We'll look at some typical continuation requests shortly.

(For the basics of **joshua:ask**: See the section "Querying the Database", page 23.)

The information resulting from the success of a backward chaining rule is not automatically added to the database, unless you explicitly do this in your program (in the continuation of the **joshua:ask**, for example).

Note that for special cases where you don't want **joshua:ask** to use backward rules, you can disable this feature by specifying **:do-backward-rules nil** to **joshua:ask**.

Here's a backward chaining example. Assuming your goal is to find a treatment for patients with a given condition, you might formulate a backward chaining rule something like this:

```
(define-predicate has-condition (person condition when))
(define-predicate effective-treatment (drug condition))
(define-predicate allergic (person drug))
(define-predicate appropriate-treatment (drug person condition))

(defrule find-cure (:backward)
  if [and [has-condition ?person ?condition today]
          [effective-treatment ?drug ?condition]
          [not [allergic ?person ?drug]]]
  then [appropriate-treatment ?drug ?person ?condition])
```

If you then **joshua:ask** what drug is appropriate for a certain condition, in a certain patient, Joshua first searches the database, after which the query triggers rule `find-cure` by matching its *then*-part. When the rule fires, Joshua works backwards from this goal to satisfy each of the component parts of the rule.

- Checks that the patient you specified has that condition (or, if you gave a variable for the patient, it iterates over all patients with that illness);
- Tries to find an effective drug for that condition; if successful, checks whether the patient is allergic to this drug.
- If the patient *is* allergic to this drug, or if Joshua simply doesn't know, it discards the drug and repeats the process of searching for another effective drug and testing for patient allergy, until it finds a drug to which the patient is not allergic, or fails trying.

For example, assume the database contains the following predications:

```
(tell [and [has-condition primadonna sore-throat today]
           [effective-treatment gargle-with-ammonia sore-throat]
           [not [allergic primadonna gargle-with-ammonia]]])
```

When you ask the system for a treatment for sore throat for all patients with that affliction, the backward rule `find-cure` triggers, satisfies its subgoals, and executes the continuation which in this case uses the convenience function **joshua:print-query** to print out the answer.

```
  (ask [appropriate-treatment ?drug ?person sore-throat] #'print-query)
[APPROPRIATE-TREATMENT GARGLE-WITH-AMMONIA PRIMADONNA SORE-THROAT]
```

5.1.2.1. Explaining Backward Chaining Support

As mentioned earlier, **joshua:ask** passes its continuation a list containing the answer it found (or derived) together with information tracing the reasoning process that was followed (for example, whether the answer was found in the database or came from backward rules, and if so, which one(s), and so on).

Joshua provides two convenience functions that extract and interpret the support information for you. We've introduced these earlier. (See the section "Querying the Database", page 23.) We mention them again here, since their usefulness becomes far more apparent in a rule context.

joshua:print-query-results extracts and displays the successful query and tells you why it succeeded.

This, for example, is the reasoning Joshua went through to answer our query about the right treatment for sore throat.

```
(ask [appropriate-treatment ?drug ?person sore-throat] #'print-query-results)
[APPROPRIATE-TREATMENT GARGLE-WITH-AMMONIA PRIMADONNA SORE-THROAT] succeeded
  It was derived from rule FIND-CURE
  [HAS-CONDITION PRIMADONNA SORE-THROAT TODAY] succeeded
    [HAS-CONDITION PRIMADONNA SORE-THROAT TODAY] was true in the database
  [EFFECTIVE-TREATMENT GARGLE-WITH-AMMONIA SORE-THROAT] succeeded
    [EFFECTIVE-TREATMENT GARGLE-WITH-AMMONIA SORE-THROAT] was true in the database
  [not [ALLERGIC PRIMADONNA GARGLE-WITH-AMMONIA]] succeeded
    [ALLERGIC PRIMADONNA GARGLE-WITH-AMMONIA] was false in the database
```

This tells us that the query pattern triggered rule `find-cure`, and that each of this rule's three subgoals was satisfied from a database lookup. Since backward chaining stops when it reaches database predications, that is the end of the support information; there is no attempt to trace why those database predications are valid.

If more than one backward rule had been invoked to find the answer, **joshua:print-query-results** would have interpreted that information as well, tracing the support through each rule to the predications used to satisfy parts of the rule.

The convenience function **joshua:graph-query-results** gives you the same information as **joshua:print-query-results** but in graph form.

Figure 5 shows what this graph looks like for our previous query.

The top of the graph shows the satisfied query and names the rule that satisfied it. Ovals denote queries that were *not* satisfied in the database. Rectangles denote queries (in this case subgoals) that were satisfied from the database. The arrows point from the support to the object being supported.

In our example, the graph shows the database predications that satisfied the rule's subgoals. Note that the Database heading indicates when the truth value for a given predication is **joshua:*false***.

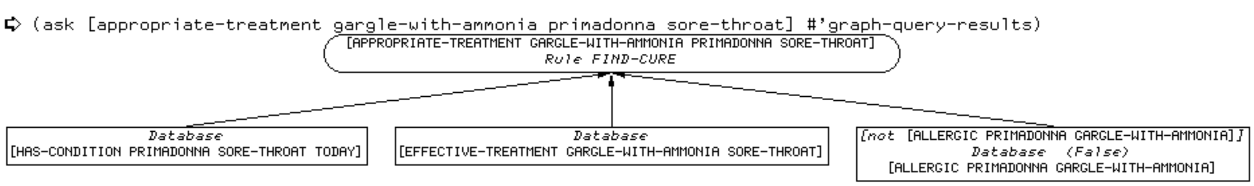


Figure 5. Graphing query support from backward rule

If you prefer to extract and interpret the support information yourself rather than have it interpreted for you, Joshua provides *accessor functions* to let you do this. Consult the dictionary entry for **joshua:ask** to see how these accessor functions work.

(If you have a TMS, you can get an explanation of forward support, also. That is, with a TMS Joshua can tell you the reasons why a database predication is in the database. The function **joshua:explain** displays the support in a manner similar to the output of **joshua:print-query-results**; **joshua:graph-tms-support** graphs the explanation in a manner similar to **joshua:graph-query-results**.)

You can trace backward rule operation by using the tracing facility. See the section "Tracing Rules", page 50.

5.2. Removing Joshua Rule Definitions

You can remove rule definitions from the system either individually or collectively.

The function **joshua:undefrule** removes a single rule definition.

For example:

```
(undefrule 'danger-sign)
```

You can do this same operation from a Zmacs editor buffer with the extended Zmacs command m-X Kill Definition. For a sample interaction with this command: See the macro **joshua:undefine-predicate**, page 152. Note that any predications previously deduced by a rule still remain in the database after you remove the rule's definition.

To remove all rule definitions at once, use the function **joshua:clear**. We have been using this function to clear the database, but it has a second optional argument to let you clear all rule definitions.

The full argument list of **joshua:clear** is:

```
CLEAR: (&OPTIONAL (CLEAR-DATABASE T) UNDEFRULE-RULES)
```

Thus, typing `(clear t t)` clears the database *and* at the same time removes all rule definitions. Be aware that this is a rather drastic step, as applications depending on these rules *will no longer work*, until you reload these rules.

The command Clear Joshua Database has an option to let you clear all Joshua rules. The cautions just mentioned apply here as well.

5.3. Tracing Rules

Commands:	Enable Joshua Tracing Disable Joshua Tracing

While debugging Joshua programs you often want to be able to watch the rules execute. You can trace the behavior of rules by using the command

```
Enable Joshua tracing (type of tracing) Forward Rules
```

for forward rules, or the command

```
Enable Joshua Tracing (type of tracing) Backward Rules
```

for backward rules.

This causes a trace message to be printed every time a rule runs. With forward rules the message appears each time the *if*-part of the rule (the trigger) is successfully completed, and the *then*-part (the action) is about to be executed. Here's a simple example of tracing forward rules.

Example:

```
(define-predicate higher-in-food-chain (eater lower-in-food-chain))
(define-predicate favorite-meal (eater food))

(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])
   (cp:execute-command "Show Joshua Database"))
```

```
Command: Enable Joshua Tracing (Type of tracing) Forward Rules
```

```
Foward Chaining tracing is on
   Tracing All forward rules
   Traced events: Fire and Queue
```

```
Command: (meals)
 ▶ Firing forward rule BASIC-FOOD-CHAIN (1 trigger)
 ▶ Firing forward rule BASIC-FOOD-CHAIN (1 trigger)
   ▶ Firing forward rule TRANSITIVE-FOOD-CHAIN (2 triggers)
 ▶ Firing forward rule TRANSITIVE-FOOD-CHAIN (2 triggers)
 ▶ Firing forward rule TRANSITIVE-FOOD-CHAIN (2 triggers)
 ▶ Firing forward rule BASIC-FOOD-CHAIN (1 trigger)

    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
```

Notice that as one rule firing **joshua:tell**s a predication that causes another rule to fire, the tracing facility indents another level. This shows you the dependency between the rules. Also, various items in the trace display are mouse sensitive and can provide more information about the program execution.

Tracing backward rules is a little more complicated, as they can be used to generate multiple solutions to a query. There are four events associated with the running of a backward rule.

First, when we try to match the trigger of the rule (the *then*-part of a backward rule), the trace message says that we are **Trying** the rule. When we successfully complete the rule action (the goals in the *if*-part), the message says that we are **Succeeding** from the rule. As we try to find another way to satisfy the rule, the message says that we are **Retrying** the rule. And lastly, when there are no more solutions for the rule, the trace message says that the rule is **Failing.**

The backward rule tracing facility uses this terminology to let you follow the execution of the rules as Joshua tries to satisfy a query. In order to demonstrate the tracing of backward rules we will first enable the tracing of both predications and backward rules. Tracing predications along with rules shows you the **joshua:ask**s or **joshua:tell**s corresponding to each rule firing. Here's a simple example of backward rule tracing.

Example:

```
(define-predicate has-condition (person condition when))
(define-predicate effective-treatment (drug condition))
(define-predicate allergic (person drug))
(define-predicate appropriate-treatment (drug person condition))

(defrule find-cure (:backward)
  if [and [has-condition ?person ?condition today]
          [effective-treatment ?drug ?condition]
          [not [allergic ?person ?drug]]]
  then [appropriate-treatment ?drug ?person ?condition])

(tell [and [has-condition primadonna sore-throat today]
           [effective-treatment gargle-with-ammonia sore-throat]
           [not [allergic primadonna gargle-with-ammonia]]])
```

```
Command: Enable Joshua Tracing (Type of tracing) Predications

Predication tracing is on
   Tracing All predicates
   Traced events: Ask and Tell
```

```
Command: Enable Joshua Tracing (Type of tracing) Backward Rules

Backward Chaining tracing is on
   Tracing All backward rules
   Traced events: Try, Fail, Retry, and Succeed
```

```
Command: (ask [appropriate-treatment ≡drug ≡person sore-throat]
        #'print-query)
 ▶ Asking predication [APPROPRIATE-TREATMENT ≡DRUG
                                  ≡PERSON SORE-THROAT]
 ▶ Trying backward rule FIND-CURE  (Goal... )
   ▶ Asking predication [HAS-CONDITION ≡PERSON
                                       SORE-THROAT TODAY]
   ▶ Asking predication [EFFECTIVE-TREATMENT ≡DRUG
                                             SORE-THROAT]
   ▶ Asking predication [NOT [ALLERGIC PRIMADONNA
                                       GARGLE-WITH-AMMONIA]]
   ▶ Asking predication [not [ALLERGIC PRIMADONNA
                                       GARGLE-WITH-AMMONIA]]
 ▶ Succeeding backward rule FIND-CURE

[APPROPRIATE-TREATMENT GARGLE-WITH-AMMONIA PRIMADONNA SORE-THROAT]
 ▶ Retrying backward rule FIND-CURE  (Goal... )
 ▶ Failing backward rule FIND-CURE
```

Notice that even after the rule derives the only possible answer and prints the unified query, backward rule tracing displays the **Retrying** and **Failing** events as the rule tries and fails to find another answer.

You can get a greater degree of control over rule tracing by providing the **:menu** keyword to Enable Joshua Tracing. This lets you specify particular rules to trace, trace only rules triggered by certain predications, and trace rule importance queuing and dequeuing.

To disable rule tracing use the command

```
Disable Joshua Tracing (type of tracing) Forward Rules
```

for forward rules, or

```
Disable Joshua Tracing (type of tracing) Backward Rules
```

for backward rules.

You can also adjust rule tracing options by mousing on rule names and predications. Mouse right on the object to show the possible options.

5.4. Joshua Rule Basics At a Glance

Figure 6, page 54 summarizes the basic rule information we've just covered.

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 6. Summary of Joshua Rule Operation

Sometimes the knowledge needed to satisfy a rule trigger can be elicited from the user through the question facility. This works somewhat similarly to backward rules, and we discuss it in the chapter that follows. See the section "Asking the User Questions", page 55.

6. Asking the User Questions

We have seen that knowledge stored in the database can be extended by reasoning about it with rules. Questions behave similarly to rules, and are another way of extending knowledge by seeking out information from the user. "User" in this context is a very general term denoting any person, process, or device that a question can interact with.

Questions are like backward chaining rules. A question has a name, a trigger pattern and a body, and you use it in goal-directed inference to satisfy backward chaining goals.

This chapter assumes that you are familiar with the basics of rule operation presented in the chapter"Rules and Inference".

joshua:ask does not automatically invoke questions, unless you specify **:do-questions t**. (In contrast, rules are used by default, unless you specify **:do-backward-rules nil**.)

Joshua searches for applicable questions *after* searching the database and running all appropriate backward rules. When a question trigger unifies with the query pattern, the question body runs, and, if successful, calls the **joshua:ask** continuation.

Joshua supplies a default question facility that you can customize if you wish.

6.1. Adding and Removing Joshua Question Definitions

joshua:defquestion defines a question. The basic arguments to **joshua:defquestion** are a rule *name*, a *control-structure* that specifies forward or backward questions (currently can only be **:backward**), and a *pattern* specifying the question's trigger.

Like rules, questions have control structure arguments **:importance** and **:documentation**. The former lets you prioritize the order in which questions are run. The latter lets you add a documentation string to explain what your question does.

In the custom version you supply a question *body*, using the keyword argument :*code* before the custom code.

Please consult the dictionary entry for **joshua:defquestion** for a full description of this macro.

The function **joshua:undefquestion** removes a single question definition. To remove a question from a Zmacs buffer, use m-X Kill Definition.

6.2. Default Joshua Questions

The basic components are a name, a **:backward** control structure, optional control structure arguments, and a trigger pattern. For example:

```
(defquestion question1 (:backward)
        [author-of Winnie-the-Pooh Milne])

(defquestion question2 (:backward
                        :documentation "Get information about authorship")
        [author-of ?work ?author])
```

Here's what happens if the question is invoked:

- If the query contains no logic variables at run time, a Yes or No question is generated once. (A No answer means the proposition is false or unknown.)

- If the query does contain logic variables at run time, the question loops, presenting iterations of an AVV (Accept Variable Values) menu, each looking for values of the variables that would make the triggers true.

Either the default **joshua:say** method or a user-written **joshua:say** method if available is used in formatting the question.

Here are some examples:

```
(define-predicate lost (person object))
(define-predicate in-possession-of (object suspect))
(define-predicate suspects (person spouse))
(define-predicate unfaithful (person to-spouse))

(defquestion where-is-it? (:backward)
        [lost ?person ?object])
```

Now we do an **joshua:ask**, specifying that we want questions to be run.

```
Example 1: No variables in the query that triggers the question
  (ask [lost Desdemona handkerchief] #'print-query :do-questions t)
Is it true that "[LOST DESDEMONA HANDKERCHIEF]"?  [default No]: Yes
[LOST DESDEMONA HANDKERCHIEF]
```

Let's add a user-defined **joshua:say** method to the above example and do another **joshua:ask**.

```
Example 2:
(define-predicate-method (say lost) (&optional (stream *standard-output*))
  (with-statement-destructured (person object) self
    (format stream "~A lost the ~A" person object)))
```

```
  (ask [lost Desdemona handkerchief] #'print-query :do-questions t)
Is it true that "DESDEMONA lost the HANDKERCHIEF"?  [default No]: Yes
[LOST DESDEMONA HANDKERCHIEF]
```

In the next example, the query that triggers the question at run time contains variables. The question loops until we indicate that no more solutions exist by clicking on No and on End.

Example 3:

```
➪ (ask [lost ≡who ≡what] #'print-query :do-questions t)
For what values of ≡PERSON and ≡OBJECT is it true that "≡PERSON has lost the ≡OBJECT"?
Some solution exists: Yes  No
Value for ≡PERSON: SAMSON
Value for ≡OBJECT: HAIR
<ABORT> aborts, <END> uses these values

[LOST SAMSON HAIR]
What are some more values of ≡PERSON and ≡OBJECT such that "≡PERSON has lost the ≡OBJECT"?
Some solution exists: Yes  No
Value for ≡PERSON: ROBIN
Value for ≡OBJECT: PET-SALAMANDER
<ABORT> aborts, <END> uses these values

[LOST ROBIN PET-SALAMANDER]
What are some more values of ≡PERSON and ≡OBJECT such that "≡PERSON has lost the ≡OBJECT"?
Some solution exists: Yes  No
<ABORT> aborts, <END> uses these values
```

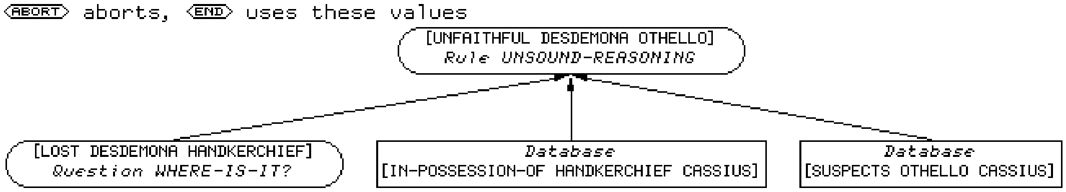
Example 4 uses information obtained from the question to satisfy a backward rule's subgoal. The rule imitates Othello's emotional logic when he concludes that Desdemona is unfaithful to him because her lost handkerchief turns up in the possession of Cassius (whom Othello suspects of being her lover). We use the question to establish that Desdemona has indeed lost her handkerchief and thus to satisfy the rule's first subgoal.

Here's the rule definition and a function that shows how to use it. To see how **joshua:ask** arrived at its answer we use the convenience function, **joshua:graph-query-results**.

Example 4:

```
(defrule fidelity-test (:backward)
  if [and [lost ?person ?object]
          [in-possession-of ?object ?a-suspect]
          [suspects ?spouse ?a-suspect]]
  then [unfaithful ?person ?spouse])

(defun desdemoniad ()
  ;; see if Desdemona is unfaithful to Othello
  (clear)
  (tell [and [in-possession-of handkerchief Cassius]
             [suspects Othello Cassius]])
  (ask [unfaithful Desdemona Othello] #'graph-query-results
       :do-questions t))
```

```
➪ (desdemoniad)
For what values of ≡OBJECT is it true that "[LOST DESDEMONA ≡OBJECT]"?
Some solution exists: Yes  No
Value for ≡OBJECT: HANDKERCHIEF
ABORT aborts, END uses these values
```




```
What are some more values of ≡OBJECT such that "[LOST DESDEMONA ≡OBJECT]" is true?
Some solution exists: Yes  No
ABORT aborts, END uses these values
```

6.3. Writing Custom Questions

To write a custom version of a question, use the basic question format adding the keyword, **:code**, and its arguments before you write the question body.

```
(defquestion <question-name> (:backward)
             <trigger-pattern>
  :code
  ((query truth-value continuation &optional query-context)
   <body>)
```

query is the query for **joshua:ask**, unified with the trigger pattern of the question.

If *truth-value* is **joshua:*true***, the system is asking whether the statement in true, as opposed to being false or unknown. If *truth-value* is **joshua:*false***, the system is asking whether the statement is **joshua:*false***, as opposed to being true or unknown.

The *query-context* argument can almost always be ignored.

The question *body* can be Lisp forms or Joshua commands, and it 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. This is how questions with variables at run-time can loop, eliciting all possible bindings from the user.

body can do anything, including an **joshua:ask** or **joshua:tell**. It should, however, 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 the continuation.

- If there are logic variables present, solicit sets of bindings for them from somewhere (for example, the user). For each such set, call **joshua:succeed**.

The question in this example tries to find one or more languages to which a given word belongs. Standard choices are offered in a languages menu. The user can click on one or more of these choices, as well as enter any others. We also ask the function **joshua:succeed** to return a user-id telling us who answered the question.

To see the results and how they came about, we use the convenience function, **joshua:print-query-results** in the **joshua:ask** continuation.

```
(define-predicate valid-word (word language))

;;; The customized defquestion
(defquestion check-if-valid-word (:backward)
   [valid-word ?word ?language]
   :code
   ((query truth-value continuation &optional ignore)
    (unless (eql truth-value *true*)
      (error "I don't know how to ask if ~S is false." query))
    (typecase ?word
      (unbound-logic-variable
        (error "I don't know how to ask questions about every
                possible word: ~S" query))
      (otherwise
        (typecase ?language
          (unbound-logic-variable
            (let ((list-of-languages
                  (dw:accepting-values
                   (*query-io* :label "Languages" :own-window t)
                    (format *query-io*
                              "Languages in which ~A is a word" ?word)
                    (append
                     (accept '(subset english french german swahili sanskrit)
                              :prompt nil)
                     (accept '(sequence symbol) :prompt "Others" :default nil)))))
            (loop for the-language in list-of-languages
                  do (with-unification
                       (unify ?language the-language)
                       (succeed sys:user-id)))))
          (otherwise
            (when (dw:accepting-values
                     (*query-io* :label "Languages" :own-window t)
                     (format *query-io*
                               "Is ~A a word in ~A?  " ?word ?language)
                     (accept 'boolean :prompt nil :prompt-mode :raw :default t))
              (succeed sys:user-id))))))))

(ask [valid-word boutique ?language] #'print-query-results :do-questions t)
```

Click on your selections, and they are highlighted. As the next screen shows, we selected two languages from the default, and added another one from the "Others" option. After we click on Done, the **joshua:ask** continuation executes, printing out the answers to the query and the reason why each answer succeeded. Since we asked **joshua:succeed** to return a user id telling us who answered the question, we get this information as well.


```
  (ask [valid-word boutique ?language] #'print-query-results :do-questions t)
[VALID-WORD BOUTIQUE FRENCH] succeeded
  It was derived from question CHECK-IF-VALID-WORD
  "Pinhead"
[VALID-WORD BOUTIQUE ENGLISH] succeeded
  It was derived from question CHECK-IF-VALID-WORD
  "Pinhead"
[VALID-WORD BOUTIQUE ESPERANTO] succeeded
  It was derived from question CHECK-IF-VALID-WORD
  "Pinhead"
```

7. Pattern Matching in Joshua: Unification

Functions	**joshua:unify** **joshua:variant** **joshua:succeed**
Macro	**joshua:with-unification**

As described in the section "Rules and Inference", pattern matching underlies all inferencing operations in Joshua. In forward chaining, Joshua matches rule trigger patterns with database predications. In backward chaining, it matches goals with database predications and with rule trigger patterns. The type of pattern matching used is called *unification*.

7.1. Unification Rules

Two predications that contain no variables match (or unify), if they are structurally *equivalent*, that is, if they "look the same". This is much the same test as the Lisp **equal** function. For example:

```
[fact a b c]  matches         [fact a b c]
[fact a b c]  does not match  [fact a b d]
```

The more interesting case for pattern matching is when the predication patterns contain logic variables. Then predications unify if there is a way of *substituting values for the variables* so that both predications become structurally equivalent. In the simple case, an uninstantiated logic variable matches any object in the equivalent position, becoming instantiated as that object. In the pattern pairs below, for example, these are the matches and substitutions :

```
Pattern 1: [fact a b c]   }
Pattern 2: [fact a b ?x]  }
                 fact = fact (fact matches fact)
                 a = a       (a matches a)
                 b = b       (b matches b)
                 ?x --> c    (c is substituted for ?x)


Pattern 1: [fact a b]     }
Pattern 2: [fact ?x ?y]   }
                 fact = fact (fact matches fact)
                 ?x --> a;
                 ?y --> b;
```

Advanced Concepts Note:

The "occur-check" states that it is not valid to substitute for a variable a structure containing that variable. One reason for this is that such a substitution might cause the system to draw logically unsound inferences.

The following do not unify:

```
[f ?x ?x]

[f (g ?y) ?y]
```

For a detailed discussion of the "occur-check": See the function **joshua:unify**, page 154.

To determine the correct match in the next example, one needs to know when variables with the same name are identical and when they differ. For instance, why does a trigger pattern like this:

```
[f ?x ?x]
```

match the first predication pattern below, but not the second one.

```
[f goo goo]      ;matches trigger pattern
[f silly putty]  ;does not match trigger pattern
```

Section "Variables and Scoping in Joshua" covers these scoping rules.

7.2. Variables and Scoping in Joshua

In general terms, the scope of a variable is the area within which it is visible and can be referred to. For more on scoping within Symbolics Common Lisp: See the section "Scoping" in *Symbolics Common Lisp Language Concepts*.

In Joshua, since logic variables typically have names (?x, ?item, and so on), valid matching is based on understanding when variables with the same name are identical and when they differ. The scoping rules for Joshua determine this.

Logic variables are *lexically scoped* within rule bodies. That is, *logic variables with the same name within a rule body are identical and must match the same object.*

```
Trigger pattern using ?x:        [f ?x 1]  ;this ?x is different from
Database predication using ?x:   [f 2 ?x]  ;this ?x

Matches and substitutions:

In trigger     [f ?x 1]   ?x --> 2  becomes    [f 2 1]
In predication [f 2 ?x]   ?x --> 1  becomes    [f 2 1]
```

Used by itself, the question-mark or the equivalence symbol (?) is an anonymous logic variable. Each one you type is different, imposing no scoping constraints, and can therefore be used for "no care" slots. For example:

```
          [foo ? ?]    matches   [foo 1 2]
     and  [foo ≡ ≡]    matches   [foo 1 2]
 whereas  [foo ?x ?x]  does not
```

Logic variable names are new each time a rule is triggered. Conceptually the system makes a copy of the triggered rule. The variable names in the "copy" are the same names as those in the original, but are a different version of these variables. During rule execution this "copy" of the rule is successively modified as new unifications occur. Once rule execution terminates, all bindings are undone and the rule "copy" is discarded.

7.3. Some Examples of Joshua Unification

Here are some simple examples:

```
[father ?x ?y]      unifies with   [father john john-jr]
[father ?x john]    unifies with   [father george ?y]
[father ?x ?y]      unifies with   [father ?z ?z]
[father ?x john]    does not unify with    [father ?y george]
```

Here's a forward rule example with a **format** statement to help us see what is happening.

```
(define-predicate plays-instrument (person instrument))
(define-predicate owns (person thing))
(define-predicate invite-to-audition (person))

(tell [plays-instrument Jane tuba])

(defrule test-eligibility (:forward)
  if [and [plays-instrument ?person ?instrument]
          [owns ?person ?instrument]]
  then [and [invite-to-audition ?person]
            (format t "~% In the rule body ?person is bound to ~A, and ?instrument
 is bound to ~A." ?person ?instrument)])
```

When this rule's trigger pattern (`[plays-instrument ?person ?instrument]`) is matched against the database, it unifies with `[plays-instrument Jane tuba]`. That is, `?person` is unified with `Jane`, and `?instrument` is unified with `tuba`.

If we add another **joshua:tell** statement such as `[owns Jane tuba]`, this unifies with the second part of the rule's trigger, and the rule fires. The action part then adds the rule's inference to the database with the bindings that have been established by unification. The format message we added to the action part confirms these bindings:

```
    (tell [owns Jane tuba])
   In the rule body ?person is bound to JANE, and ?instrument
   is bound to TUBA.
  [OWNS JANE TUBA]
```

The next example uses backward chaining and compound subgoals.

Backward rule `ownership-1` states that a person owns whatever object s/he has paid for. Backward rule `ownership-2` states that if Fred and Bob own the same item then it is a desirable item. We then **joshua:ask** a query whose goal is to determine what items are desirable. The query succeeds. What items unified?

The query `[desirable item]` unifies with the trigger of backward rule `ownership-2`; that rule's subgoal `[owns ?person ?object]`, then unifies with the trigger of rule `ownership-1`. The latter's subgoal `[paid-for ?person ?object]` unifies with the database predication `[paid-for Fred stereo]`. The goal of rule `ownership-1` is satisfied. Next Joshua tries to satisfy the second part of rule `ownership-2`'s subgoal, `[owns Bob ?item]` and so on, until the goal succeeds.

Here are the definitions and **joshua:tell** statements for this example.

```
;;; Define some additional predicates
(define-predicate paid-for (person thing))
(define-predicate desirable (thing))

(tell [and [paid-for Fred stereo]
           [paid-for Bob  stereo]])

(defrule ownership-1 (:backward)
  if [paid-for ?person ?object]
  then [owns ?person ?object])

(defrule ownership-2 (:backward)
  if [and [owns Fred ?item]
          [owns Bob ?item]]
  then [desirable ?item])
```

If we could look inside the unifier during execution of the query, this is what we would see:

```
(ask [desirable ?x] #'print-query)
```

```
Asking [DESIRABLE ?X]
Firing backward rule OWNERSHIP-2
|      Unifying [DESIRABLE ?X] with [DESIRABLE ?ITEM]
|      Unifying ?ITEM with ?X
| Asking [OWNS FRED ?X]
| Firing backward rule OWNERSHIP-1
| |      Unifying [OWNS FRED ?X] with [OWNS ?PERSON ?OBJECT]
| |      Unifying ?PERSON with FRED
| |      Unifying ?OBJECT with ?X
| | Asking [PAID-FOR FRED ?X]
| |      Unifying ?X with STEREO
| | Asking [OWNS BOB STEREO]
| Firing backward rule OWNERSHIP-1
| |      Unifying [OWNS BOB STEREO] with [OWNS ?PERSON ?OBJECT]
| |      Unifying ?PERSON with BOB
| |      Unifying ?OBJECT with STEREO
| | Asking [PAID-FOR BOB STEREO]
[DESIRABLE STEREO]
```

7.4. Basic Unification Facilities

Joshua provides some unification facilities for use within Lisp code. The dictionary entry for each facility details its use and provides examples.

The function **joshua:unify** unifies expressions within Lisp code embedded in the *if*-part of rules, or in the body of a **joshua:defquestion**, or wherever you find it convenient to call it yourself.

The macro **joshua:with-unification** establishes the scope of unifications done within its body, and establishes a place to be thrown to if a unification in its body fails. That is:

- If a unification cannot be done, **joshua:unify throw**s to the dynamically innermost enclosing **joshua:with-unification**, and

- The extent of the unification is the dynamic extent of the dynamically innermost enclosing **joshua:with-unification**.

The function **joshua:succeed** will, based on its context, find the continuation and call it accordingly.

The function **joshua:variant** is related to **joshua:unify**, but shouldn't be confused with it. Whereas **joshua:unify** tries to see if two objects can be *made* the same, **joshua:variant** checks whether two objects *are* the same. Predications that differ only in the names of the logic variables they contain are equivalent, and are *variants* of each other. (**joshua:tell** uses **joshua:variant** to check whether the predication it is adding to the database is already there.)

joshua:variant is based on the notion that it should not matter what the names of the logic variables are, so long as the structures are the same. This is a much stronger condition than **joshua:unify**. Pairs that satisfy **joshua:unify** are not necessarily variants, but every pair that satisfies **joshua:variant** also satisfies **joshua:unify**.

8. Using Joshua Within Lisp Code

Here are several basic functions that you can use inside Lisp code. These functions can be grouped under predication facilities and unification facilities. The tables below summarize each function's use. Please consult the respective dictionary entries for more detail and examples.

Predication Facilities:

joshua:different-objects	If the arguments are **joshua::eql**, or if either argument is an uninstantiated logic variable, **joshua:different-objects** returns **nil**. Otherwise it returns **t**.
joshua:make-predication	Constructs a predication (does not enter it into the database). The [ ] syntax is a reader macro that expands into this.
joshua:predication	The base flavor for all predications in Joshua. This works well with functions like **joshua::typep** and **joshua::typecase**.
joshua:predicationp	Checks whether an object is built on **joshua:predication**.
joshua:with-statement-destructured	Lets you bind Lisp variables to predication arguments.

Unification Facilities:

joshua:succeed	Based on context, finds and calls continuation
joshua:unify	Unifies two Joshua patterns, while side-effecting any logic variables for the extent of the enclosing **joshua:with-unification**
joshua:variant	Checks whether two predications are equivalent, differing only in the names of the logic variables they contain. This does not need to be done under a **joshua:with-unification**; it merely returns **nil** if it fails.
joshua:with-unification	Establishes the scope within which substitutions specified by **joshua:unify** take effect; establishes a place to be thrown to if a unification in its body fails.

9. 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 92.

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.

10. Justification and Truth Maintenance

Functions:	**joshua:explain** **joshua:unjustify** **joshua:graph-tms-support**
Keyword:	**:justification**
Symbols:	**:premise** **:assumption** **:none**

A Truth Maintenance System (TMS) is a tool used by deductive systems to keep track of interdependencies among statements in a database.

A TMS has two main functions:

- Recording and maintaining the reasoning that supports the current set of predications in the database
- Maintaining the logical consistency of these predications

Since very often database predications logically depend on each other as determined by rules, keeping track of these dependency relationships lets the system explain its reasoning process. Moreover, knowing the dependencies for each predication lets the system work out the consequences of changes to the truth values of predications and modify the database to keep it current and consistent. Database modifications involve *retracting* (removing) facts. This means that a justification is a bidirectional link, from fact to conclusion *and back*.

Since a TMS may not always be necessary, its inclusion is optional in Joshua. Without a TMS, Joshua records the truth value of each predication, and changes this value if new **joshua:tell** statements assert a different truth value. See the section "Truth Values", page 20.

However, without a TMS, the system has no knowledge of the reasons supporting its beliefs, and hence no awareness of logical contradictions. Once the database becomes inconsistent it remains so unless you modify it under program control. On the other hand, you don't pay either the space or time penalties of having a TMS. The tradeoff is up to you.

An inconsistent database may or may not be acceptable depending on the problem. If you do want to use a TMS, Joshua currently supports a clausal TMS (LTMS) that you can mix into your predicate definitions. This TMS is based on David McAllester's three-valued TMS (Massachussetts Institute of Technology, Artificial Intelligence Lab, A.I. Memo 473, 5 May, 1978).

This chapter is an overview of how Joshua works with the LTMS.

To include a TMS in your application, you specify the particular TMS model you are using (LTMS, JTMS, any other) as an argument to your predicate definition. For instance:

```
(define-predicate temperature-of (object temperature)
                  (ltms:ltms-predicate-model))
```

Here, as in the following examples, we use the supplied LTMS model, which resides in its own package.

Advanced Concepts Note:

You can incorporate any TMS of your choice into your Joshua application. As with any other tool you want to build into Joshua this is straightforward, since Joshua talks to Truth Maintenance Systems via a generic protocol; thus you need only write protocol methods for the TMS generic functions. Joshua's facilities for supporting external TMS systems are discussed in the *Joshua Reference Manual*.

With a TMS model Joshua can *justify*, *explain*, and *revise* its beliefs. We explore each of these activities in the following sections.

10.1. Justification

Joshua *believes* a fact to be valid if it is in the database and has a truth value of **joshua:*true*** or **joshua:*false***. (A truth value of **joshua:*unknown*** denotes a fact whose validity is currently not known.) Without a TMS the system records truth values, but not the reasons for them.

See the section "Truth Values", page 20.

When a TMS is present, a believed predication must have a *justification*. This means there must be at least one currently valid reason supporting the predication's truth status. When **joshua:tell** enters a new predication it provides a justification for it. The TMS records this justification as part of the information it maintains about the database predication. Truth values and their justification status can change as a result of new information and resulting action taken by the TMS, or as a result of program action.

Because the LTMS ensures data consistency, it does not accept logically contradictory facts; thus it cannot believe simultaneously predication `[P ...]` and predication `[NOT [P ...]]` (that is, a predication cannot be **joshua:*true*** and **joshua:*false*** at the same time). If this happens, one of these statements must be retracted by removing its justification (**joshua:unjustify**).

TMS justifications can be either *primitive*, or *compound*. The terms *primitive* and *compound* refer to the degree of reasoning used in generating the justification.

10.1.1. Primitive Justifications

Primitive justifications are primitive with respect to the reasoning process. (The system remembers no further explanation.) Primitive justifications are usually specified by the programmer or by system defaults in a **joshua:tell** used outside a rule. This type of justification which depends on nothing except itself is the basic component of the belief structure.

There are three types of primitive justification:

- *Assumptions*
- *Premises*
- *None*

Assumptions justify predications whose retraction you are willing to leave to the TMS.

Premises justify predications whose retraction you want to control yourself, rather than leave them to the TMS.

As we shall see, when the TMS finds a logical contradiction that can be traced back to a single assumption, it retracts that assumption automatically. If the inconsistent predication depends only on premises, or if there is more than one assumption in the support, the LTMS signals a condition and requests you to take care of correcting the inconsistency. That usually means using the Debugger; programs can handle the condition to do otherwise. For more on condition-handling: See the section "Conditions" in *Symbolics Common Lisp Programming Constructs*.

None justifications are justifications that don't really cause the predication to be believed. That is,

```
(tell [foo ?x] :justification :none)
```

puts the predication in the datbase, but keeps its truth value at **joshua:*unknown***. (You might want to do this because you are using **joshua:tell** to canonicalize the predication, but you don't yet want the system to believe it.)

When the LTMS is given a justification for a predication in the database, the LTMS builds a *clause* consisting of the predication and its support.

If a new predication is being added by a **joshua:tell** outside a rule context and if you supply no explicit justification, the default justification is *premise*. To see this, enable the tracing of TMS operations for the next examples.

```
Enable Joshua Tracing (Type of tracing) TMS Operations
```

```
TMS tracing is on
   Tracing All TMS predicates
   Traced events: Contradiction, Justify, and Unjustify
```

```
(define-predicate temperature-of (object temperature)
                        (ltms:ltms-predicate-model))
```

Now **joshua:tell** a fact about the temperature of some object. The trace shows that the new predication by default is being established in the database as a premise.

```
(tell [temperature-of skin cool])
 ▶ Justifying: [TEMPERATURE-OF SKIN COOL] <-- PREMISE
[TEMPERATURE-OF SKIN COOL]
T
```

To specify a justification yourself, use the keyword **:justification** within a **joshua:tell** statement, followed by the justification (a symbol or a list). If the justification is a symbol, a primitive justification is built whose mnemonic is the symbol. **:assumption** is the symbol for the primitive justification assumption. **:premise** is the symbol for the primitive justification premise, and **:none** is the symbol for the primitive justification none.

```
(tell [temperature-of surface very-hot]
      :justification :assumption)
 ▶ Justifying: [TEMPERATURE-OF SURFACE VERY-HOT] <-- ASSUMPTION
[TEMPERATURE-OF SURFACE VERY-HOT]
T
▬
```

10.1.2. Compound Justifications

There are several types of compound justification. Here we deal with the most basic type, namely, *deduction*. We discuss others in the *Joshua Reference Manual*.

When Joshua **joshua:tell**s the database a new predication inferred from a forward chaining rule, the default justification is *deduction*; this states that the fact depends on the executing rule and the predications that triggered it. These predications may in turn be deductions from other rules; ultimately the chain of dependencies leads to primitive support (premises and/or assumptions).

Joshua can extract the primitive and compound justification for a database predication and interpret it for you. See the section "Explaining Program Beliefs", page 75.

10.1.3. Database Predications Can Have Multiple Justifications

Every time you (or the program) **joshua:tell** a statement, it is given a new justification. So although there is only one version of the statement in the database, it can have multiple justifications. For example, earlier we added the predication `[temperature-of surface very-hot]` to the database as an *assumption*. Now we add a **joshua:tell** statement that triggers the forward rule `determine-temp1`.

```
(define-predicate condition-of (object condition) (ltms:ltms-predicate-model))

(tell [condition-of surface melting])

(defrule determine-temp1 (:forward)
  if [condition-of ?object melting]
  then [temperature-of ?object very-hot])
```

When this rule fires, it infers `[temperature-of surface very-hot]`. Since this predication is already in the database, the system does not reinsert it. Rather, it adds a new justification for it that mentions the rule `determine-temp1`. Note, however, that the "support" is always the current justification; the system only uses the new justification if the current one for some reason becomes obsolete. For example, if you remove the current justification, the next justification, if any, becomes the predication's support. The predication's truth value does not become **joshua:*unknown*** until every one of its justifications has been removed. For more on removing a predication's support:

See the section "Retracting Predications with **joshua:unjustify**", page 84.

10.2. Explaining Program Beliefs

The function **joshua:explain** provides information about the justification(s) for a database predication. **joshua:explain** is the forward chaining dual to **joshua:ask** continuation functions such as **joshua:graph-query-results** and **joshua:print-query-results** that trace backward chaining support for a satisfied query. (The LTMS does keep some notes about the results of backward chaining, namely the facts in the database from which you have chained.)

Here's a forward rule, `love-medieval-style`, to help us deduce who might properly engage in a courtly love affair. One of the facts that triggers this rule, the predication `[is-attached ...]`, is deduced by the forward rule `engaged`.

```
(define-predicate is-engaged (lady husband) (ltms:ltms-predicate-model))
(define-predicate is-attached (lady husband) (ltms:ltms-predicate-model))

(defrule engaged (:forward)
  if [is-engaged lady husband]
  then [is-attached lady husband])

(define-predicate noble-lady (lady) (ltms:ltms-predicate-model))
(define-predicate outranks (lady gentleman) (ltms:ltms-predicate-model))
(define-predicate in-proximity (lady gentleman) (ltms:ltms-predicate-model))
(define-predicate good-to-accept-as-lover (lady gentleman)
                      (ltms:ltms-predicate-model))

(defun courtly-setup ()
  (clear)
  (tell [and [noble-lady Isolde]
             [outranks Isolde Tristan]
             [in-proximity Isolde Tristan]
             [is-engaged Isolde Mark]]))

(courtly-setup)
```

```
(defrule love-medieval-style (:forward)
  if [and [noble-lady ?lady]
          [outranks ?lady ?gentleman]
          [in-proximity ?lady ?gentleman]
          [is-attached ?lady ?husband]
          (different-objects ?gentleman ?husband)]
  then [good-to-accept-as-lover ?lady ?gentleman])
```

Now we ask what the system knows about acceptable lover candidates, and why. **joshua:explain** must be given the actual database predication in order to extract its justification. One simple way to do this is to use the convenience function, **joshua:map-over-database-predications**.

```
(map-over-database-predications [good-to-accept-as-lover ?x ?y] #'explain)
[GOOD-TO-ACCEPT-AS-LOVER ISOLDE TRISTAN] is true
  It was derived from rule LOVE-MEDIEVAL-STYLE
  [NOBLE-LADY ISOLDE] is true
    It is a :PREMISE
  [OUTRANKS ISOLDE TRISTAN] is true
    It is a :PREMISE
  [IN-PROXIMITY ISOLDE TRISTAN] is true
    It is a :PREMISE
  [IS-ATTACHED ISOLDE MARK] is true
    It was derived from rule ENGAGED
    [IS-ENGAGED ISOLDE MARK] is true
      It is a :PREMISE
```

joshua:explain extracts and displays the chain of support through forward rules to primitive support (premises and assumptions) together with their truth values.

To see the same information in graph form, use the function **joshua:graph-tms-support**. (This is analogous to the function **joshua:graph-query-results** that graphs backward support.)

Figure 7 uses the previous example and shows the graph of the justification.

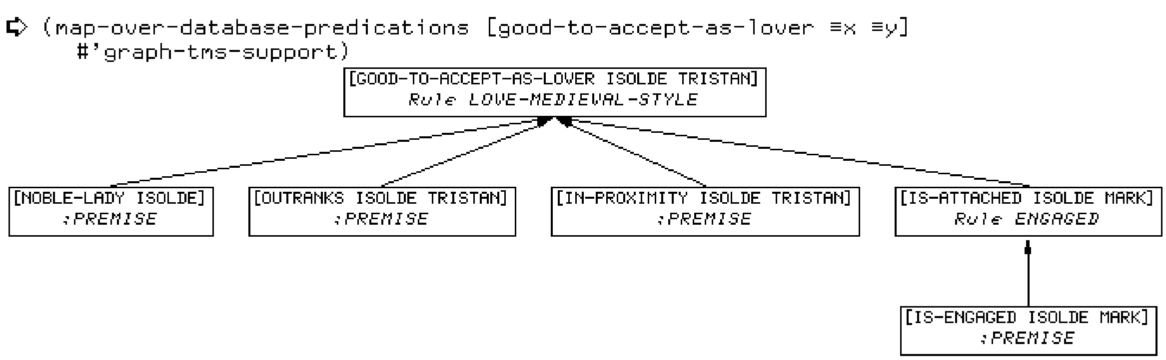


Figure 7. Sample Graph of TMS Support

Since all objects in the graph are in the database, all are drawn inside rectangles.

The top of the graph shows the database predication whose justification is being traced. The graph traces the justification through rules to underlying premises. The arrows move up from the primitive support through intermediate support to the object being explained.

Justifying and explaining the reasons underlying current knowledge are not static activities, since the flow of new information makes the database subject to constant modification. The next section surveys the role played by the TMS in keeping the knowledge base updated and logically consistent. See the section "Revising Program Beliefs", page 77.

10.3. Revising Program Beliefs

To reason effectively, a program must be able to revise beliefs when new knowledge contradicts them. A contradiction comes about when a valid justification for accepting a new belief conflicts with the justification for believing an already existing predication. A truth value of **joshua:*contradictory*** is temporarily assigned to the new predication; this alerts the TMS to correct the contradiction. Since contradictory facts cannot be simultaneously accepted, one of them must be retracted by having its justification(s) removed (**joshua:unjustify**). The TMS keeps current information consistent with new information by allowing the updating of current knowledge in accord with new reasons.

You can let the TMS handle contradictions, or you can bind into your program your own code to handle these conditions. For more on the signalling and handling of conditions: See the section "Conditions" in *Symbolics Common Lisp Programming Constructs*.

This is what happens in the absence of bound condition handlers: when the TMS detects a contradictory predication (in the case of an LTMS, a clause that cannot be satisfied) it traces backward through the reasons for the conflicting belief and finds the primitive support underlying it.

If a single assumption is causing the contradiction, the TMS automatically does an **joshua:unjustify** operation on it, and continues processing. **joshua:unjustify** removes the current support for the predication. If the predication had a single support, **joshua:unjustify** changes its truth value to **joshua:*unknown***. If there were auxiliary justifications, they may change the truth value from **joshua:*unknown*** back to either **joshua:*true*** or **joshua:*false***. (See the section "Retracting Predications with **joshua:unjustify**", page 84.)

If more than one assumption underlies the contradiction, or if the contradiction rests on premises rather than assumptions, the TMS signals an appropriate condition; if no program has bound a handler for this condition the TMS invokes the Debugger, listing all the asumptions and premises on which the contradiction depends.

The Debugger offers you the choice of retracting some justifications or of aborting out. After you make your retraction(s), processing continues until the current operation succeeds, or the TMS finds another contradiction.

10.3.1. An LTMS Example

Following is a sequence of examples showing the changes to a database as the processing of new **joshua:tell** statements gives rise to contradictions and subsequent retractions. (For simplicity, we assume a single justification for each predication.) The first example, shown in figure 8, illustrates a contradiction resulting from the attempt to enter a new fact into the database. Circled numbers in the figure indicate the sequence of events.

We've added step-by-step comments as well as Trace and Debugger displays to show interaction with the system. The comments are numbered to correspond to the figure numbers.

The definitions and **joshua:tell** statements for this example appear at the end of this section. See the section "Definitions for the LTMS Example", page 83.

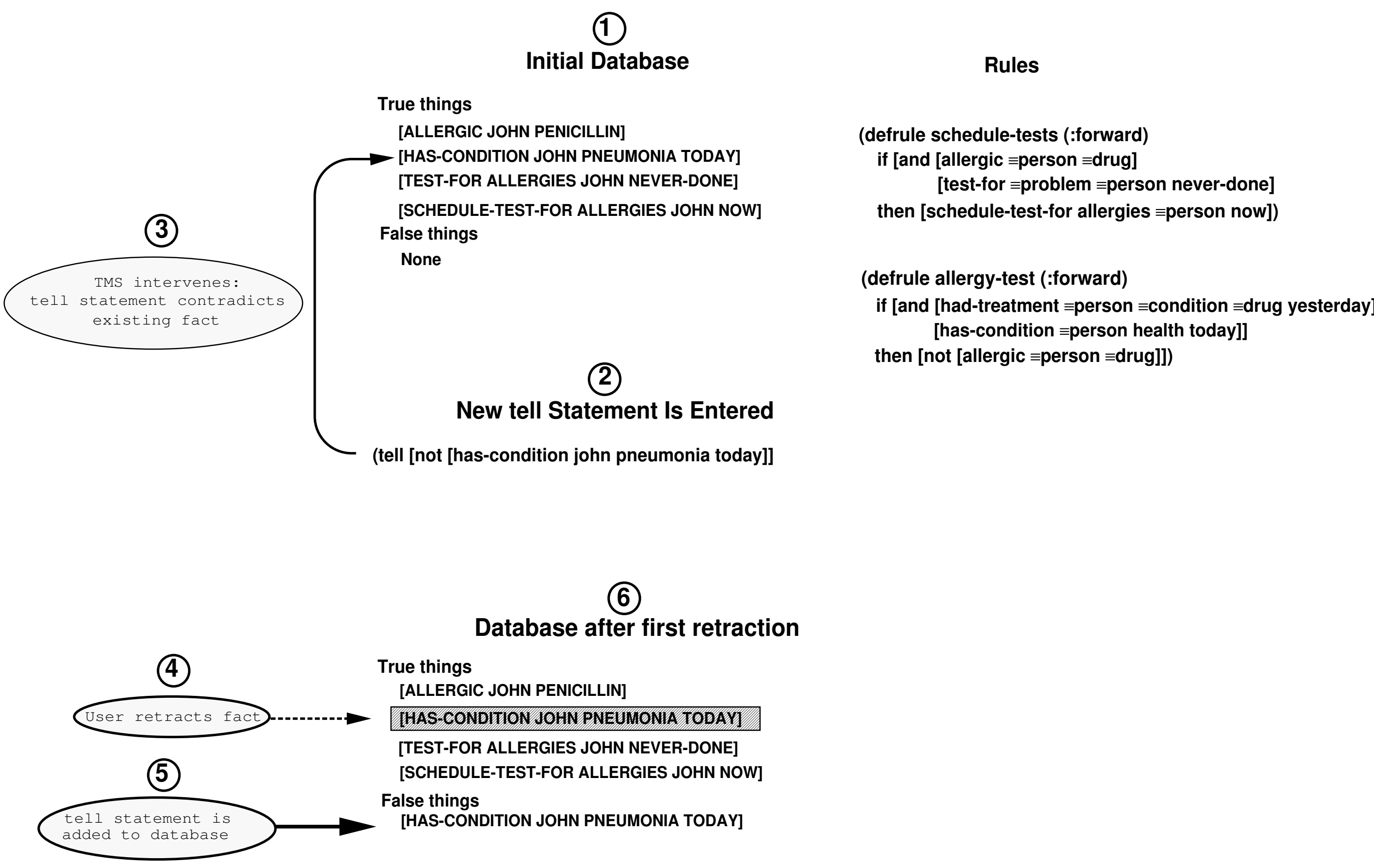


Figure 8. TMS Example -- New Fact Contradicts Existing Fact

1. We begin with an initial database and two forward chaining rules, as shown in figure 8. To see what happens, we turn on the Trace facility for TMS operations before proceeding.

```
Enable Joshua Tracing (Type of tracing) TMS Operations
```

```
TMS tracing is on
   Tracing All TMS predicates
   Traced events: Contradiction, Justify, and Unjustify
```

2. A new **joshua:tell** statement is entered and the system attempts to insert it into the database.

3. The TMS detects a contradiction between the new **joshua:tell** statement, `[not [has-condition John pneumonia today]]`, and its opposite assertion, `[has-condition John pneumonia today]`, currently residing in the database. The contradiction rests on premises and since we have not specified how to handle the condition, the Debugger is invoked. (Without a TMS, Joshua would simply have changed the truth value of the old statement to **joshua:*false***.) Figure9 shows the Trace display and the user interaction with the system after the TMS intervenes.

```
▶ Noticing contradiction: Establishing [HAS-CONDITION JOHN PNEUMONIA TODAY] as false <-- PREMISE
Error: Backtracking because:
       [HAS-CONDITION JOHN PNEUMONIA TODAY] is contradictory but has only premises in its support
       #<:PREMISE ¬[HAS-CONDITION JOHN PNEUMONIA TODAY]> is the unsatisfiable clause.
                                                       [HAS-CONDITION JOHN PNEUMONIA TODAY]
                                                                  :PREMISE
[HAS-CONDITION JOHN PNEUMONIA TODAY]
                                                       [NOT [HAS-CONDITION JOHN PNEUMONIA TODAY]]
                                                                  :PREMISE

JI::SINGLETON-SUPPORT-IS-AUTOMATIC-UNJUSTIFY
   Arg 0 (JI::CONDITION-OBJECT): #<LTMS::LTMS-CONTRADICTION 104216665>
s-A, <RESUME>:   Pick a subset of the premises to be unjustified.
s-B, <ABORT>:    Return to Lisp Top Level in Dynamic Lisp Listener 1
s-C:             Restart process Dynamic Lisp Listener 1
→ Resume Proceed
Pick a subset of the premises to be unjustified.
Subset to be unjustified: #<:PREMISE [JU::HAS-CONDITION JU::JOHN JU::PNEUMONIA JU::TODAY]>
 ▶ Unjustifying: [HAS-CONDITION JOHN PNEUMONIA TODAY]
 ▶ Justifying: [HAS-CONDITION JOHN PNEUMONIA TODAY] as false <-- PREMISE
```

Figure 9. TMS Example -- Trace and Debugger Displays, and First Retraction

4. We decide to **joshua:unjustify** the belief that John does have pneumonia. Note that we do so by pressing the `RESUME` key and clicking on those leaves of the support graph that we want to **joshua:unjustify**. The Trace display in figure9 shows that the truth value of the statement `[has-condition John pneumonia today]` is now changed from **joshua:*true*** to **joshua:*unknown***.

5. The removal of the original statement allows the system to insert the new **joshua:tell** statement, `[not [has-condition John pneumonia today]]`, into the database.

6. The database with one former statement retracted, and one new statement added under the heading "False things".

Continuing from this situation, the next example illustrates a contradiction generated by the firing of a forward chaining rule. Figure 10 shows what happens.

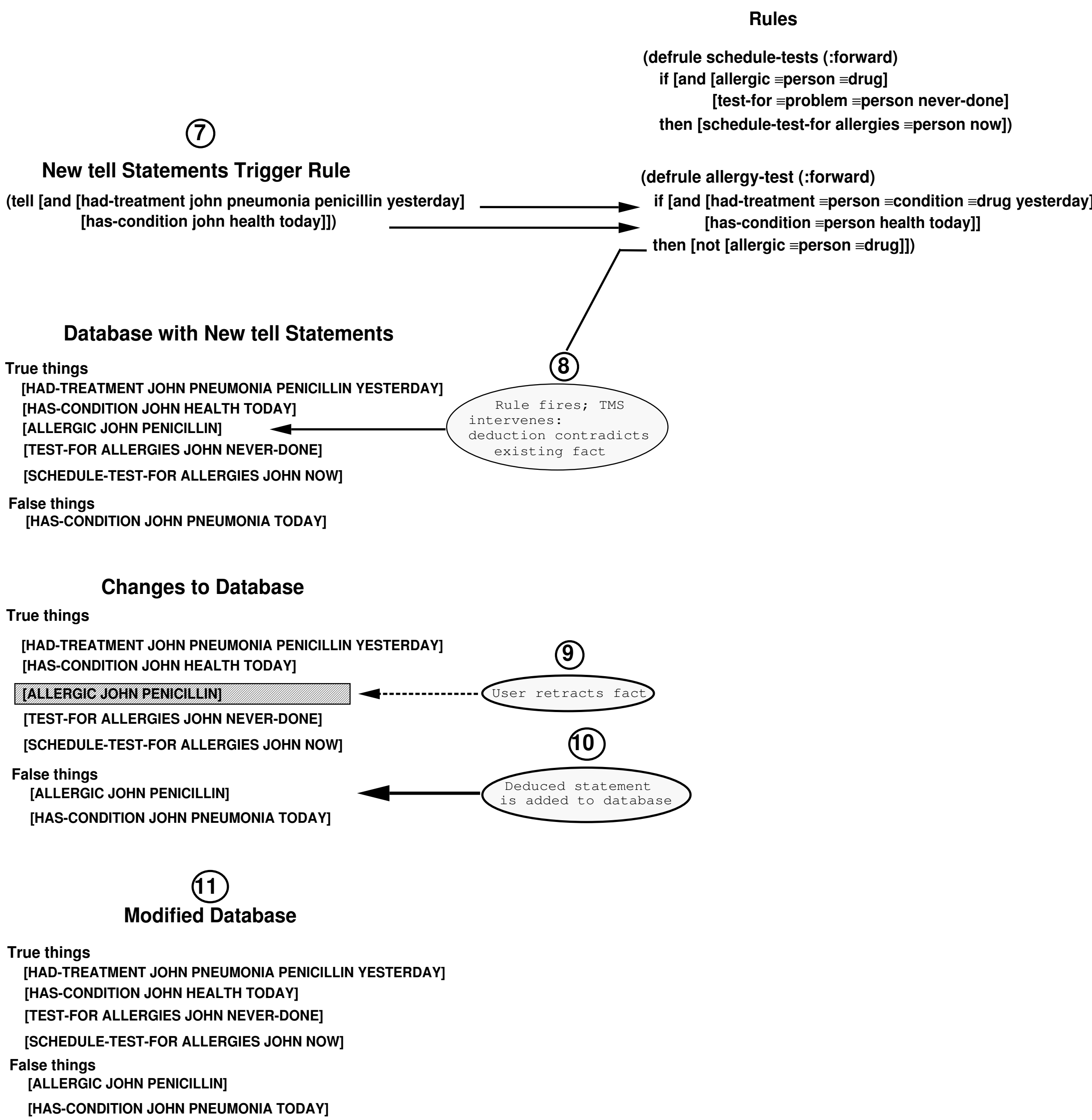


Figure 10. TMS Example, Continued -- Deduced Fact Contradicts Existing Fact

7. A compound **joshua:tell** statement of two predications joined by **joshua::and** is added to the database. These statements trigger the forward rule, `allergy-test`.

8. Rule allergy-test fires and attempts to add a newly deduced fact, [not [allergic John penicillin]], to the database. Once again, there is a contradiction; the database has a belief that John is allergic to penicillin, while the rule deduces that he is not. (Evidently, someone performed the experiment of giving John penicillin despite John's known allergy to it and John is still alive, thus posing a problem for the TMS.)

9. Bowing to the facts, we ask the system to **joshua:unjustify** the belief that [allergic John penicillin].

   Figure 11 shows the Trace display as the new statements are added to the database, the user interaction with the system after the TMS intervenes, and the Trace display showing the retraction. [We have artificially broken up the system displays into separate figures to make each step easier to follow. In real life, the entire sequence happens in one continuous event until all contradictions arising from a **joshua:tell** statement are resolved.]

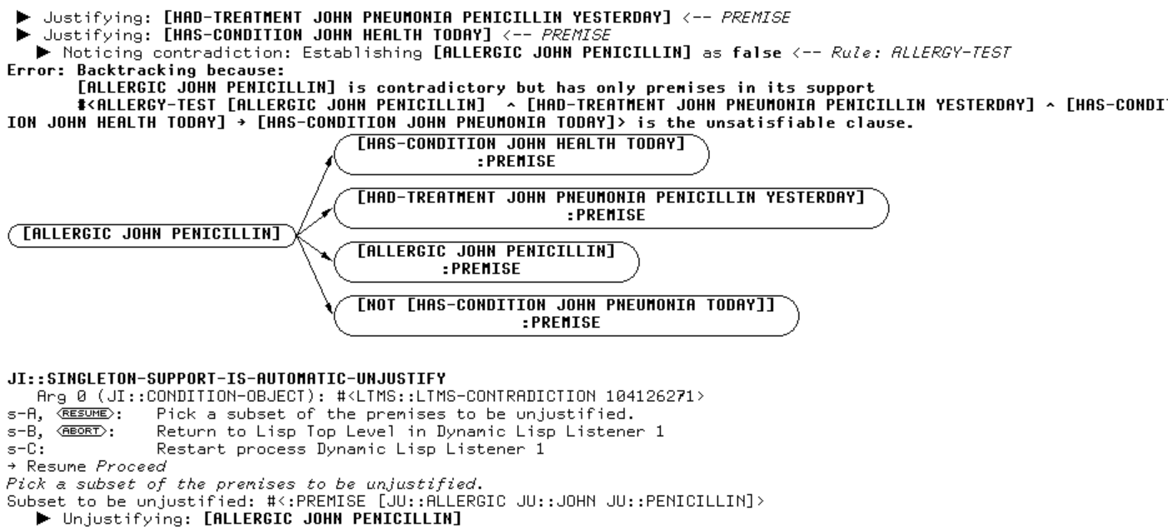


Figure 11. TMS Example, Continued -- Trace Display and Second Retraction

   As the Trace display at the bottom of the figure shows, the truth value of the predication [allergic John penicillin] that we have retracted is changed from **joshua:*true*** to **joshua:*unknown***.

10. The belief deduced by rule allergy-test, namely, [not [allergic John penicillin]], is now inserted into the database.

11. This is the modified database.

   Continuing from this point, the final sequence shows how retractions can cause inconsistencies in the database, and how the TMS automatically corrects these. Figure 12 illustrates.

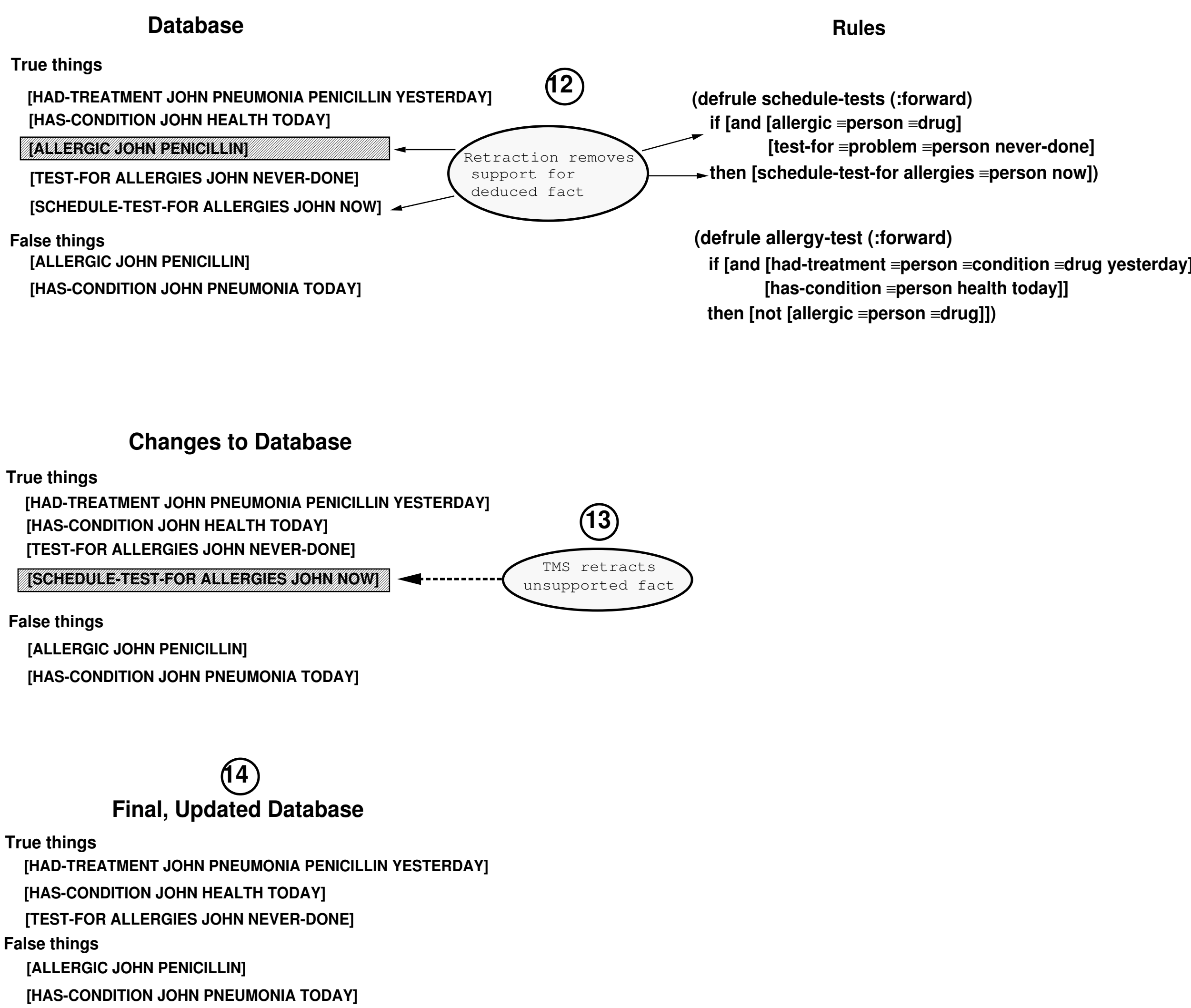


Figure 12. TMS Example, Concluded -- TMS Automatically Retracts Unsupported Fact

12. One predication in the database, namely, `[schedule-test-for-allergies John now]` depended for its support on a predication `[allergic John penicillin]` that we retracted in step 9. Without support, a predication can no longer be believed.

13. The TMS knows about all dependencies, and automatically retracts the unsupported predication `[schedule-test-for-allergies John now]`. The Trace display in Figure 13, shows this retraction.

14. This shows the final, updated, and consistent database.

```
▶ Unjustifying: [SCHEDULE-TEST-FOR ALLERGIES JOHN NOW]
▶ Justifying: [ALLERGIC JOHN PENICILLIN] as false <-- NOGOOD
[AND [HAD-TREATMENT JOHN PNEUMONIA PENICILLIN YESTERDAY] [HAS-CONDITION JOHN HEALTH TODAY]]
```

Figure 13. TMS Example, Concluded -- Automatic Retraction by the TMS

10.3.1.1. Definitions for the LTMS Example

```
;;; Predicate definitions
(define-predicate allergic (person drug)
  (ltms:ltms-predicate-model))
(define-predicate had-treatment (person condition drug when)
  (ltms:ltms-predicate-model))
(define-predicate test-for (problem person when-done)
  (ltms:ltms-predicate-model))
(define-predicate schedule-test-for (problem person when)
  (ltms:ltms-predicate-model))
(define-predicate has-condition (person condition when)
  (ltms:ltms-predicate-model))

;;; Rule definitions
(defrule schedule-tests (:forward)
  if [and [allergic ?person ?drug]
          [test-for ?problem ?person never-done]]
  then [schedule-test-for allergies ?person now])

(defrule allergy-test (:forward)
    if [and [had-treatment ?person ?condition ?drug yesterday]
            [not [has-condition ?person ?condition today]]
            [has-condition ?person health today]]
    then [not [allergic ?person ?drug]])

;;; tell statements to set up starting database
(defun setup-initial-conditions ()
  (clear)
  (tell [and [allergic john penicillin]
             [has-condition john pneumonia today]
             [test-for allergies john never-done]]))

;;; Create first contradiction
(defun create-contradiction ()
  (tell [not [has-condition john pneumonia today]]))

;;; Create second contradiction
(defun create-contradiction2 ()
  (tell [and [had-treatment John pneumonia penicillin yesterday]
             [has-condition john health today]]))
```

10.3.2. Retracting Predications with joshua:unjustify

You can use the function **joshua:unjustify** to remove the support for specific predications from the database independently of action by the TMS. (Without a TMS, **joshua:unjustify** just sets the truth value of the predication to **joshua:*unknown***.)

Note that when more than one justification supports a predication, **joshua:unjustify** must be called once for each justification. (Like **joshua:explain**, **joshua:unjustify** needs the actual database predication that you want to operate on, not a copy of it that you type in.)

In an earlier example we used a predication with two justifications. See the section "Database Predications Can Have Multiple Justifications", page 74. First we inserted this predication, `[temperature-of surface very-hot]`, into the database as an assumption. Later it was deduced from forward rule `determine-temp1` shown below, thus receiving an additional justification as a deduction.

```
(define-predicate condition-of (object condition) (ltms:ltms-predicate-model))
(define-predicate temperature-of (object temperature) (ltms:ltms-predicate-model))

(tell [condition-of surface melting])

(defrule determine-temp1 (:forward)
  if [condition-of ?object melting]
  then [temperature-of ?object very-hot])
```

Let's enable the tracing of TMS operations to see what happens when we try to **joshua:unjustify** this predication. (We get at the predication by clicking on it in the database display.)

```
  Show Joshua Database
True things
  [TEMPERATURE-OF SURFACE VERY-HOT]
  [CONDITION-OF SURFACE MELTING]
False things
  None
```

```
(unjustify [TEMPERATURE-OF SURFACE VERY-HOT])
 ▶ Unjustifying: [TEMPERATURE-OF SURFACE VERY-HOT]
 ▶ Justifying: [TEMPERATURE-OF SURFACE VERY-HOT] <-- Rule:
 DETERMINE-TEMP1
NIL
```

Although **joshua:unjustify** removed the current support for the predication, the Trace display shows that the predication is still **joshua:*true***, because it was supported by an additional justification which has now become its current support. The support says that the predication was deduced from a forward rule. This deduction remains valid as long as the predications supporting the rule's conclusion are valid. Thus, trying to **joshua:unjustify** this deduced predication now would cause an error (with the LTMS you can only **joshua:unjustify** primitive support).

```
(unjustify [TEMPERATURE-OF SURFACE VERY-HOT])
Error: Predication [TEMPERATURE-OF SURFACE VERY-HOT] can't be unjustified,
 its support is #<DETERMINE-TEMP1
 [CONDITION-OF SURFACE MELTING]  → [TEMPERATURE-OF SURFACE VERY-HOT]>
.
.
.
```

This section concludes our survey of basic Joshua concepts. You now have a working knowledge of Joshua that enables you to build applications using the default Joshua facilities.

The Joshua functions and commands covered in this manual appear in the "Basic Joshua Dictionary" immediately following this section.

Advanced Joshua concepts are covered in the companion volume to this. See the document *Joshua Reference Manual*.

11. Dictionary Notes: Basic Joshua Dictionary

The entries in this dictionary are a *subset* of those in the "Joshua Language Dictionary". We have included here only those functions and commands that you will find useful for getting started and whose operation has been discussed in the conceptual portions of this manual. Advanced functions and functions needed for modeling are omitted.

Here is the alphabetized list of Joshua language objects included in this dictionary.

11.1. List of Entries in the Basic Joshua Dictionary

joshua:defquestion
joshua:defrule

joshua:different-objects
"Disable Joshua Tracing Command"
"Enable Joshua Tracing Command"
"Explain Predication Command"
joshua:explain
joshua:*false*
joshua:graph-query-results
joshua:graph-tms-support
joshua:known
joshua:make-predication
joshua:map-over-database-predications
joshua:predication
joshua:predicationp
joshua:print-query
joshua:print-query-results
joshua:provable
"Reset Joshua Tracing Command"
"˜\\Say\\"
joshua:say
joshua:say-query
"Show Joshua Predicates Command"
"Show Joshua Rules Command"
"Show Joshua Tracing Command"
"Show Rule Definition Command"
joshua:succeed
joshua:tell
joshua:*true*
joshua:undefine-predicate
joshua:undefquestion
joshua:undefrule
joshua:unify
joshua:unjustify
joshua:*unknown*
joshua:untell
joshua:variant
joshua:with-statement-destructured
joshua:with-unbound-logic-variables
joshua:with-unification

12. Basic Joshua Dictionary

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", page 62.

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" in *Joshua Reference Manual*.

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", page 23. See the section "The Joshua Database Protocol" in *Joshua Reference Manual*. See the section "Customizing the Data Index" in *Joshua Reference Manual*.

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 91.

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 91.

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 91.

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 91.

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", page 17.

See the section "The Joshua Database Protocol" in *Joshua Reference Manual*.

See the section "Customizing the Data Index" in *Joshua Reference Manual*.

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", page 20. See the section "Justification and Truth Maintenance", page 71.

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: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 arguments 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", page 11.

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", page 41.)

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", page 55.

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", page 41.

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" in *Joshua Reference Manual*.

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" in *Joshua Reference Manual*.

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", page 41. See the section "The Joshua Rule Facilities " in *Joshua Reference Manual*.

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", page 67.

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"

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", page 37. See the section "Tracing Rules", page 50.

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", page 75.

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", page 20.

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])
```

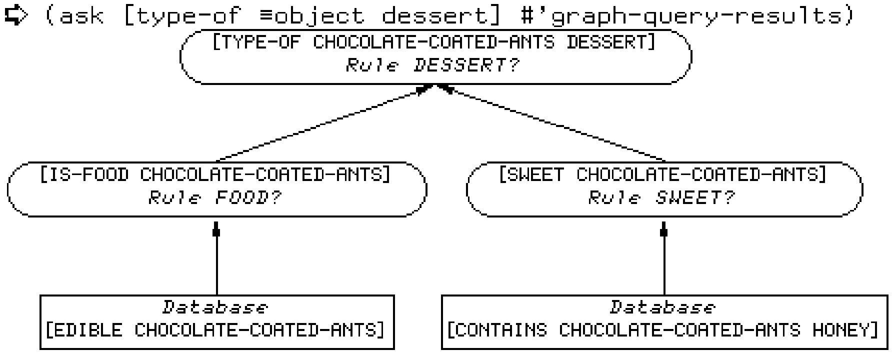
Now we **joshua:ask** what foods qualify as desserts and why. In the graph, ovals denote queries that were *not* satisfied directly by the database. Rectangles denote queries that were satisfied by the database.

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.

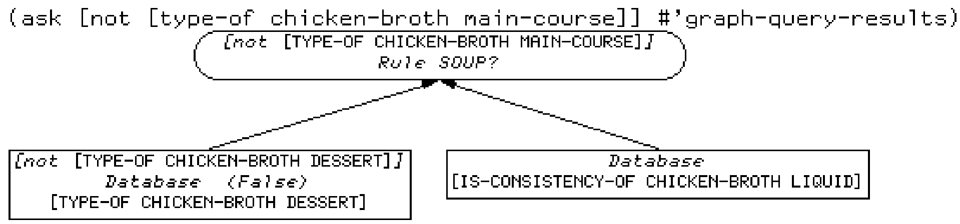
```
⇨ (ask [type-of ≡object dessert] #'graph-query-results)
```



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", page 23. See the section "Explaining Backward Chaining Support", page 48.

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
```

```
➪ (graph-tms-support [NATIVE-SPEAKER-OF SPANISH VIOLET])
```


```
NIL
```

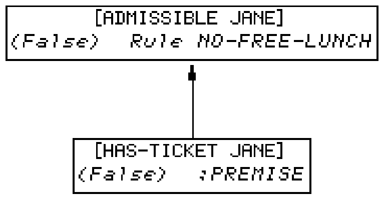
You must give **joshua:graph-tms-support** the actual predication object that resides in the database, rather than a copy of it. In our example we retrieve the predication object by clicking the mouse over it in the database display.

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])
```




```
NIL
➪
```

Predications with a truth value of **joshua:*false*** appear with an indication that they are false.

See the section "Explaining Program Beliefs", page 75.

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: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", page 11.

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", page 23.

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: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: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", page 23.

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", page 23. See the section "Explaining Backward Chaining Support", page 48.

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"

~\\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", page 35.

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:

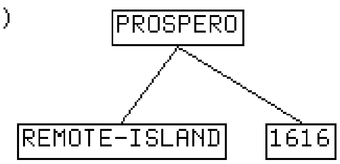
```
PROSPERO was an owner of a frobozz in 1616 at REMOTE-ISLAND.
```

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", page 35.

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", page 23.

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", page 15.

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: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" in *Joshua Reference Manual*.

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", page 15.

See the section "The Joshua Database Protocol" in *Joshua Reference Manual*.

See the section "Customizing the Data Index" in *Joshua Reference Manual*.

See the section "Truth Maintenance Facilities" in *Joshua Reference Manual*.

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", page 20.

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.

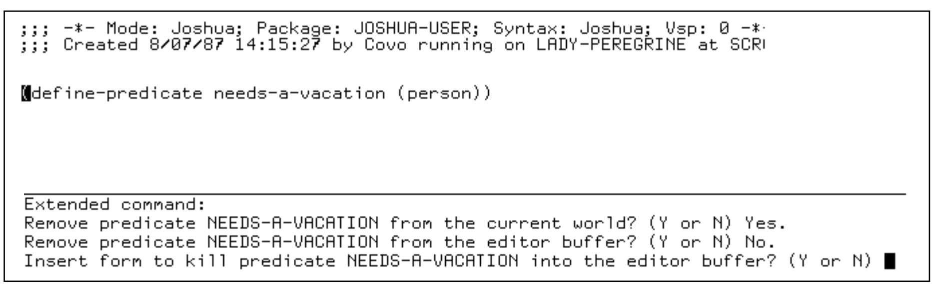
Example:

```
(define-predicate fruit (a-fruit))
(undefine-predicate 'fruit)
```

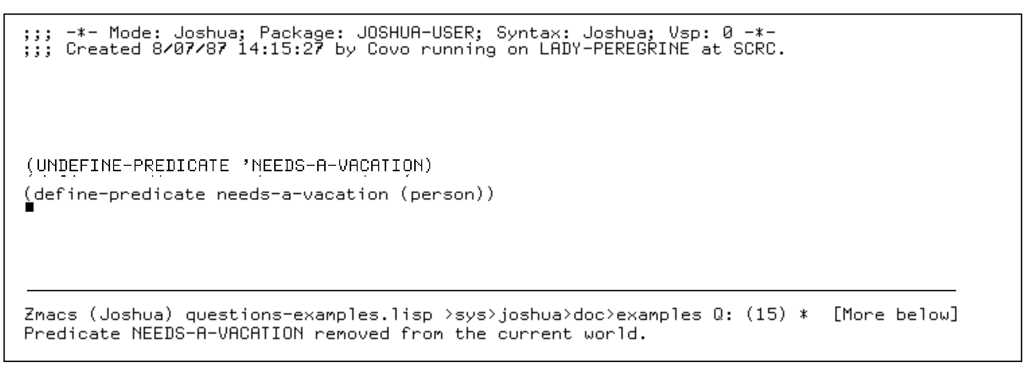
You can perform the same operation from the Zmacs editor. Place your cursor on the predicate definition to be removed and use the command m-X Kill Definition. The system asks for confirmation in the minibuffer; then it offers you the options of removing the definition from the editor buffer itself, and of inserting the **joshua:undefine-predicate** command into the editor buffer.

Example:

1. Interaction During m-X Kill Definition



2. Zmacs Buffer After Completion of m-X Kill Definition



Related Functions:

joshua:define-predicate
"Zmacs Command: Kill Definition"

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 152.

Related Functions:

joshua:defquestion
"Zmacs Command: Kill Definition"

See the section "Asking the User Questions", page 55.

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 152.

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" in *Joshua Reference Manual*.

Related Functions:

joshua:defrule
joshua:clear
"Clear Joshua Database Command"
"Zmacs Command: Kill Definition"

See the section "Rules and Inference", page 41.

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", page 61.

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 159. **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", page 77.

See the section "Retracting Predications with **joshua:unjustify**", page 84.

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", page 20.

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 believed (**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", page 17.

See the section "The Joshua Database Protocol" in *Joshua Reference Manual*.

See the section "Customizing the Data Index" in *Joshua Reference Manual*.

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", page 62.

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:

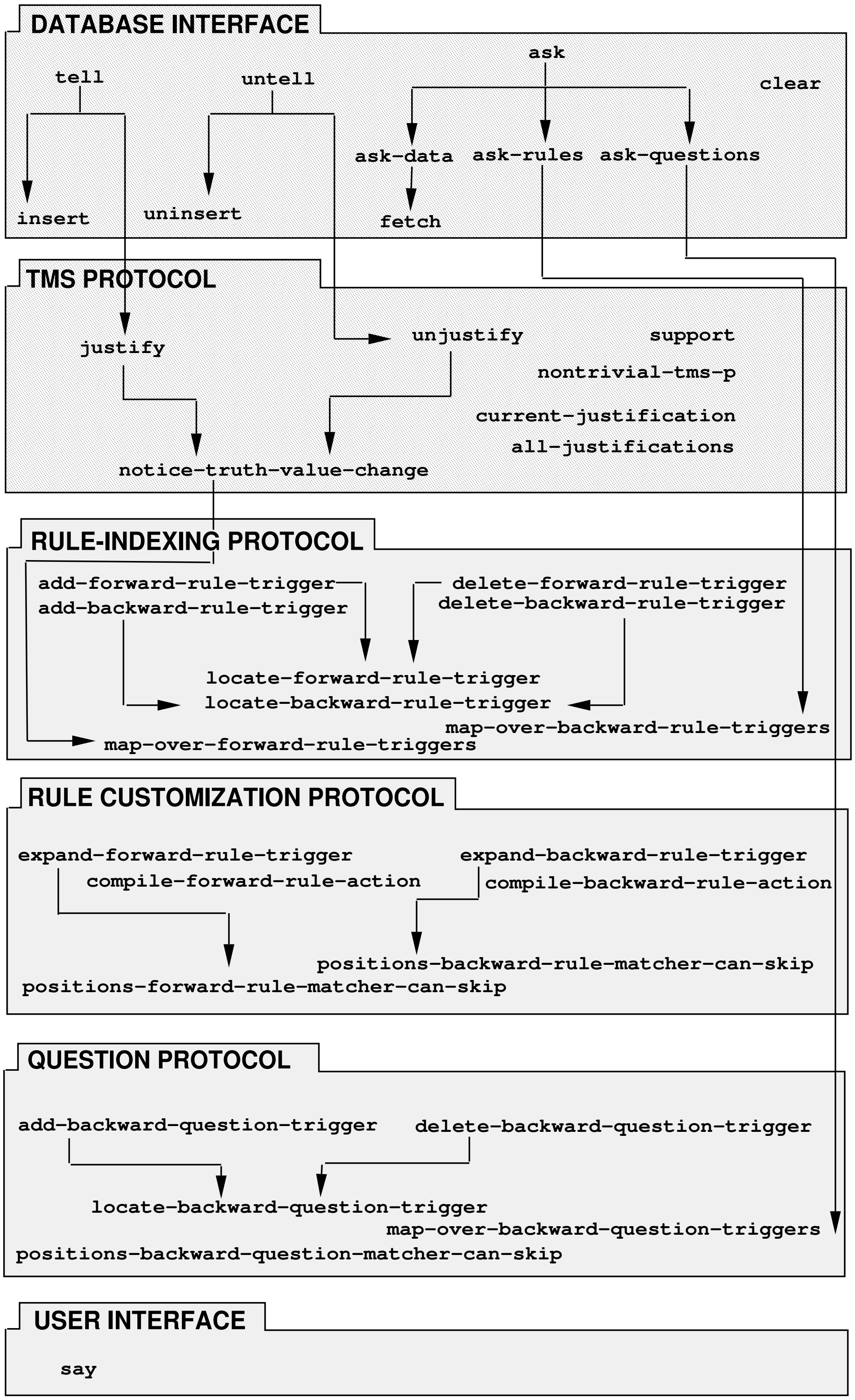
```
(define-predicate candidate-word (a-word))
(define-predicate is-computer-jargon (some-word))
(defvar *computer-jargon* '(foo bar baz quux))


(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", page 61.

Appendix A
A Figure of the Joshua Protocol of Inference


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
RULE-INDEXING PROTOCOL
add-forward-rule-trigger
add-backward-rule-trigger
delete-forward-rule-trigger
delete-backward-rule-trigger
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
compile-forward-rule-action
expand-backward-rule-trigger
compile-backward-rule-action
positions-backward-rule-matcher-can-skip
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
positions-backward-question-matcher-can-skip
USER INTERFACE
say

Index