Copyright and Trademarks

KnowledgeWorks and Prolog User Guide (Unix version)

Version 8.0

December 2021

Copyright © 2021 by LispWorks Ltd.

All Rights Reserved. No part of this publication may be reproduced, stored in a retrieval system, or transmitted, in any form or by any means, electronic, mechanical, photocopying, recording, or otherwise, without the prior written permission of LispWorks Ltd.

The information in this publication is provided for information only, is subject to change without notice, and should not be construed as a commitment by LispWorks Ltd. LispWorks Ltd assumes no responsibility or liability for any errors or inaccuracies that may appear in this publication. The software described in this book is furnished under license and may only be used or copied in accordance with the terms of that license.

LispWorks and KnowledgeWorks are registered trademarks of LispWorks Ltd.

Adobe and PostScript are registered trademarks of Adobe Systems Incorporated. Other brand or product names are the registered trademarks or trademarks of their respective holders.


The XP Pretty Printer bears the following copyright notice, which applies to the parts of LispWorks derived therefrom: Copyright © 1989 by the Massachusetts Institute of Technology, Cambridge, Massachusetts. Permission to use, copy, modify, and distribute this software and its documentation for any purpose and without fee is hereby granted, provided that this copyright and permission notice appear in all copies and supporting documentation, and that the name of M.I.T. not be used in advertising or publicity pertaining to distribution of the software without specific, written prior permission. M.I.T. makes no representation about the suitability of this software for any purpose. It is provided "as is" without express or implied warranty. M.I.T. disclaims all warranties with regard to this software, including all implied warranties of merchantability and fitness. In no event shall M.I.T. be liable for any special, indirect or consequential damages or any damages whatsoever resulting from loss of use, data or profits, whether in an action of contract, negligence or other tortious action, arising out of or in connection with the use or performance of this software.

LispWorks contains part of ICU software obtained from http://source.icu-project.org and which bears the following copyright and permission notice:

ICU License - ICU 1.8.1 and later
COPYRIGHT AND PERMISSION NOTICE
Copyright © 1995-2006 International Business Machines Corporation and others. All rights reserved.
Permission is hereby granted, free of charge, to any person obtaining a copy of this software and associated documentation files (the "Software"), to deal in the Software without restriction, including without limitation the rights to use, copy, modify, merge, publish, distribute, and/or sell copies of the Software, and to permit persons to whom the Software is furnished to do so, provided that the above copyright notice(s) and this permission notice appear in all copies of the Software and that both the above copyright notice(s) and this permission notice appear in supporting documentation.
THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT OF THIRD PARTY RIGHTS. IN NO EVENT SHALL THE COPYRIGHT HOLDER OR HOLDERS INCLUDED IN THIS NOTICE BE LIABLE FOR ANY CLAIM, OR ANY SPECIAL INDIRECT OR CONSEQUENTIAL DAMAGES, OR ANY DAMAGES WHATSOEVER RESULTING FROM LOSS OF USE, DATA OR PROFITS, WHETHER IN AN ACTION OF CONTRACT, NEGLIGENCE OR OTHER TORTIOUS ACTION, ARISING OUT OF OR IN CONNECTION WITH THE USE OR PERFORMANCE OF THIS SOFTWARE.
Copyright and Trademarks

Except as contained in this notice, the name of a copyright holder shall not be used in advertising or otherwise to promote the sale, use or other dealings in this Software without prior written authorization of the copyright holder. All trademarks and registered trademarks mentioned herein are the property of their respective owners.

US Government Restricted Rights
The LispWorks Software is a commercial computer software program developed at private expense and is provided with restricted rights. The LispWorks Software may not be used, reproduced, or disclosed by the Government except as set forth in the accompanying End User License Agreement and as provided in DFARS 227.7202-1(a), 227.7202-3(a) (1995), FAR 12.212(a)(1995), FAR 52.227-19, and/or FAR 52.227-14 Alt III, as applicable. Rights reserved under the copyright laws of the United States.

<table>
<thead>
<tr>
<th>Address</th>
<th>Telephone</th>
<th>Fax</th>
</tr>
</thead>
<tbody>
<tr>
<td>LispWorks Ltd</td>
<td>From North America:</td>
<td>From North America:</td>
</tr>
<tr>
<td>St. John's Innovation Centre</td>
<td>877 759 8839 (toll-free)</td>
<td>617 812 8283</td>
</tr>
<tr>
<td>Cowley Road</td>
<td>From elsewhere:</td>
<td>From elsewhere:</td>
</tr>
<tr>
<td>Cambridge</td>
<td>+44 1223 421860</td>
<td>+44 870 2206189</td>
</tr>
<tr>
<td>CB4 0WS</td>
<td></td>
<td></td>
</tr>
<tr>
<td>England</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

www.liseworks.com
# Contents

1 Introduction 8  
1.1 KnowledgeWorks 8  
1.2 Notation and conventions 10  

2 Tutorial 11  
2.1 Getting Started 11  
2.2 Loading the Tutorial 11  
2.3 Running the Tutorial 12  
2.4 Browsers 13  
2.5 KnowledgeWorks Listener 17  
2.6 Debugging 18  
2.7 Lisp Integration 20  
2.8 Systems 21  
2.9 Exiting KnowledgeWorks 22  

3 Rules 23  
3.1 Forward chaining 23  
3.2 Backward Chaining 27  
3.3 Common Lisp Interface 29  

4 Objects 30  
4.1 CLOS objects 30  
4.2 Relational Database Objects 31  
4.3 KnowledgeWorks Structures 36  

5 The Programming Environment 37  
5.1 The KnowledgeWorks Listener 37  
5.2 The Editor 38  
5.3 Clearing KnowledgeWorks 39  
5.4 The System Browser 39  
5.5 The Class Browser 40  
5.6 The Objects Browser 42  
5.7 The Rule Browser 44  
5.8 Debugging with the Environment 45  
5.9 Monitor Windows 45
Contents

6 Advanced Topics  49
   6.1 Control Flow  49
   6.2 Optimization  54
   6.3 Use of Meta-Classes  57
   6.4 Logical Dependencies and Truth Maintenance  58
   6.5 Inferencing States  59

7 Reference Guide  61
   all-debug  61
   any  61
   assert  62
   clear-all  63
   clear-rules  64
   conflict-set  64
   context  65
   current-cycle  66
   cut  66
   *cycle*  67
   defcontext  67
   def-kb-class  69
   def-kb-struct  70
   def-named-kb-class  71
   defrule  72
   deftactic  73
   destroy-inferencing-state  74
   erase  75
   fail  75
   findall  76
   findallset  76
   find-inferencing-state  77
   fire-rule  78
   get-kb-object  79
   infer  80
   *inferencing-state*  80
   inferencing-state-name  81
   *in-interpreter*  82
   instantiation  82
   inst-bindings  83
   inst-rulename  84
   inst-token  84
   kb-name  85
   kw-class  86
   lex  87
Appendix A: Common Prolog  
A.1 Introduction  
A.2 Syntax  
A.3 Defining Relations  
A.4 Using The Logic Interpreter  
A.5 Accessing Lisp From Common Prolog  
A.6 Calling Prolog From Lisp  
A.7 Debugging  
A.8 Common Prolog Macros  
A.9 Defining Definite Clause Grammars  
A.10 Edinburgh Syntax  
A.11 Graphic Development Environment  
A.12 Built-in Predicates  
A.13 Adding Built-in Predicates  

6
## Contents

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>A.14 Edinburgh Compatibility Predicates</td>
<td>129</td>
</tr>
<tr>
<td><strong>Appendix B: Examples</strong></td>
<td>130</td>
</tr>
<tr>
<td>B.1 The Tutorial</td>
<td>130</td>
</tr>
<tr>
<td>B.2 Explanation Facility</td>
<td>132</td>
</tr>
<tr>
<td>B.3 Uncertain Reasoning Facility</td>
<td>136</td>
</tr>
<tr>
<td>B.4 Other Examples</td>
<td>139</td>
</tr>
<tr>
<td><strong>Appendix C: Implementation Notes</strong></td>
<td>140</td>
</tr>
<tr>
<td>C.1 Forward Chainer</td>
<td>140</td>
</tr>
<tr>
<td>C.2 Backward Chainer</td>
<td>140</td>
</tr>
<tr>
<td><strong>Appendix D: For More Information</strong></td>
<td>142</td>
</tr>
<tr>
<td>D.1 General References</td>
<td>142</td>
</tr>
<tr>
<td>D.2 The LispWorks manuals</td>
<td>142</td>
</tr>
<tr>
<td><strong>Appendix E: Converting Other Systems</strong></td>
<td>144</td>
</tr>
<tr>
<td>E.1 OPS5</td>
<td>144</td>
</tr>
<tr>
<td>E.2 Prolog</td>
<td>146</td>
</tr>
<tr>
<td><strong>Glossary</strong></td>
<td>147</td>
</tr>
<tr>
<td><strong>Index</strong></td>
<td></td>
</tr>
</tbody>
</table>
1 Introduction

1.1 KnowledgeWorks

KnowledgeWorks® is a LispWorks® toolkit for building knowledge based systems. It is a multi-paradigm programming environment which allows developers to express problems in terms of objects, rules, and procedures. This section provides an historical perspective and an overview of the system.

1.1.1 Background

Broadly speaking, there have been two generations of commercial knowledge based system (KBS) shells. The first generation of KBS shells were built on top of symbolic programming languages such as Lisp. These shells exhibited a high degree of flexibility and functionality as a result, but suffered because of their lack of standardization, poor performance, and inability to communicate with other applications. The second generation of KBS shells were generally written in C to attack the latter two weaknesses of Lisp-based shells. However these C-based shells are inevitably less flexible, and exacerbate the standardization issue. Although written in a C (a standard language), each C-based shell must re-invent a range of features already provided as standard in every Common Lisp implementation, including the object-system and even elementary structures like lists.

KnowledgeWorks addresses all of these issues by providing a high performance rule-based system for LispWorks. The latter is a full and efficient Common Lisp implementation including the Common Lisp Object System (CLOS), and foreign function interfaces to languages such as C, C++, and FORTRAN. Hence KnowledgeWorks constitutes a tightly integrated multi-paradigm programming environment, allowing all the most powerful features of rule-based, object-oriented and procedural approaches to be combined without abandoning accepted standards.

1.1.2 Technical Overview

KnowledgeWorks includes:

- High performance inferencing mechanisms:
  - forward chaining (OPS compatible).
  - backward chaining (Prolog compatible).
- A powerful standard object system (CLOS).
- A flexible standard procedural language (Common Lisp).
- Metaprotocols for extending the object and rule systems (MOP & MRP — see below).
- Support for multiple independent inferencing operations using inferencing state objects.
- A full set of graphical tools for developing and debugging knowledge bases.
- Built using the CAPI and integrated with the LispWorks IDE.
- Integration within larger applications, possibly following a completely different paradigm.
1 Introduction

The parts of KnowledgeWorks

KnowledgeWorks rules perform pattern-matching directly over the object base (KnowledgeWorks CLOS objects and KnowledgeWorks structures). Forward chaining rules use this pattern-matching to perform actions, while backward chaining rules use it to deduce goals. The actions of forward chaining rules can call backward chaining rules, and the backward chaining inference engine may also invoke the forward chainer. Forward chaining rules may be grouped to increase the modularity of the rulebase and to introduce a mechanism for procedural control by explicit invocation of rule groups.

KnowledgeWorks CLOS objects are conventional CLOS objects with the simple addition of a mixin class providing KnowledgeWorks functionality, and they can be used outside the rulebase as ordinary CLOS objects. Any existing CLOS code may simply be reused and augmented with rules by adding the mixin to chosen classes.

LispWorks CLOS includes an implementation of the Meta Object Protocol (MOP) which allows the object system to be extended and customized in a standard way. In the same spirit of self-reflection, KnowledgeWorks rule-based system can be extended and customized using a Meta Rule Protocol (MRP) which allows meta-interpreters to be defined for rules. Together these protocols mean that KnowledgeWorks defines a region rather than a point in space of KBS shells, and ensure that developers are not constrained by the default behavior of the system.

KnowledgeWorks has a comprehensive programming environment that enables rapid development and debugging of rulebases. Tools are provided that enable the interactive examination of classes and objects. Graphical debugging windows allow forward and backward chaining rules to be single-stepped and monitored. The full LispWorks programming environment and tools are also available, for example, the editor which allows rules to be defined and redefined incrementally and dynamically (see the Editor User Guide). You can include KnowledgeWorks in a delivered runtime application if you have LispWorks Enterprise Edition, LispWorks for iOS Runtime or LispWorks for Android Runtime. See the Delivery User Guide for details.
1 Introduction

1.2 Notation and conventions

1.2.1 Prolog syntax

Syntax will be presented in BNF. Any other non-standard notation will be explained as used.

::= introduces a definition.
<..> token, or non-terminal symbol.
[..] delimits optional items.
* 0 or more repetitions of the previous token.
+ 1 or more repetitions of the previous token.
| separates alternatives.

1.2.2 Viewing example files

This manual sometimes refers to example files in the LispWorks library via a Lisp form like this:

(example-edit-file "kw/animal/defsystem")

These examples are Lisp source files in your LispWorks installation under lib/8-0-0-0/examples/. You can simply evaluate the given form to view the example source file.

The examples files are in a read-only directory and therefore you should compile them inside the IDE (by the Editor command Compile Buffer or the toolbar button or by choosing Buffer > Compile from the context menu), so it does not try to write a fasl file.

If you want to manipulate an example file or compile it on the disk rather than in the IDE, then you need first to copy the file elsewhere (most easily by using the Editor command Write File or by choosing File > Save As from the context menu).

1.2.3 Appearance of the graphical tools

Screenshots in this manual show the KnowledgeWorks tools running on Linux under GTK+ with a particular window manager and theme. So if you use GTK+ with a different window manager or theme, or Motif, you should expect a different appearance from that depicted here.

The screenshots in this manual show toolbars that may have been customized (using the context menu) so you might see some differences from your setup.
The tutorial is a simple example based on an animal guessing game. In this game the user thinks of an animal and the program asks yes/no questions. Eventually the program mentions an explicit animal and asks whether it is correct. If so, the game ends. If it is not correct it will ask what the animal was and ask for a question to distinguish it from its last guess. This is a trivial example of a learning program. The tutorial assumes a certain familiarity with Lisp, LispWorks and the Common Lisp Object System (CLOS).

All examples in this chapter assume that you are typing in expressions in a package that uses the KW package, for instance, Kw-USER.

2.1 Getting Started

To run the tutorial, put this form in your LispWorks initialization file (usually called .lispworks):

```
(require "kw")
```

Start LispWorks. The KnowledgeWorks Podium window will appear. Note the position of the KnowledgeWorks menu, which you will use to access the tools described in this manual.

KnowledgeWorks Podium

2.2 Loading the Tutorial
First bring up a KnowledgeWorks Listener by choosing KnowledgeWorks > Listener from the KnowledgeWorks Podium. The KnowledgeWorks Listener accepts Lisp input as well as KnowledgeWorks input. Enter:

```
(in-package "KW-USER")
```

into the KnowledgeWorks Listener, and then change the current directory to that of the animals demo by entering:

```
(cd (lispworks:example-file "kw/animal/"))
```

If this fails, check the value of the Lisp variable `*lispworks-directory*`. Load the tutorial by typing:

```
(load "defsystem")
```

to load the tutorial system definition, and:

```
(compile-system "ANIMAL" :load t :target-directory (get-temp-file))
```

to compile and load the rules and object base (CLOS objects). In interpreting these two commands, the KnowledgeWorks Listener has behaved just like a Lisp Listener. In general, whenever input has no specific KnowledgeWorks interpretation, the KnowledgeWorks Listener just accepts it as Lisp.

### 2.3 Running the Tutorial

First run the tutorial example a few times. Think of an animal and type `infer` into the listener. `infer` is a function which starts the forward chaining engine. Popup question windows will appear, which require clicking on either Yes or No. If your animal is guessed correctly, execution will terminate and the listener prompt will reappear. If the final guess is incorrect then:

1. Another popup will ask what the animal was. Type in the name of an animal and press Return (or click on OK). If the animal is already known to the system this constitutes an error. A confirmer popup will inform you of this; click on Confirm and execution will terminate.
2. You will be asked for a question to distinguish your animal from the system’s last guess. Type in a question (again without quotes or double-quotes) and press Return. Execution will terminate.

3. The tutorial may be restarted by typing \texttt{(infer)} again in the listener. This time the system will know about your new animal and the question that distinguishes it. Every time the rule interpreter finishes, it will return and display in the listener the number of rules the forward chaining engine fired.

2.4 Browsers

There are a number of browsers for examining the state of KnowledgeWorks. They will be introduced here, and again when the Programming Environment is discussed in \textbf{5 The Programming Environment}.

2.4.1 Rule Browser

KnowledgeWorks Rule Browser

This may be obtained by choosing \texttt{KnowledgeWorks > Rules}. The defined forward chaining contexts (or rule groups) are displayed in a drop-down list at the top. There is also a special pseudo-context for all the backward chaining rules, which is shown initially. In this case, the only other context is named \texttt{DEFAULT-CONTEXT}. Below that are listed the rules for the selected context. Choose \texttt{DEFAULT-CONTEXT} from the drop-down list and click on one of the rules, for example \texttt{PLAY}, and edit it by choosing \texttt{Rule > Find Source} from the menu bar. An editor window will appear showing this rule definition.

What this rule says is:

\begin{verbatim}
(root ?r node ?node)
(not (current-node ? node ?))
-->
((capi:display-message " ANIMAL GUESSING GAME -- think of an animal to continue")
(assert (current-node ? node ?node))
\end{verbatim}
which means:

If the node `?node` is the root node of the tree of questions, and there is no current node indicating the question about to be asked, then tell the user to think of an animal and make the root node `?node` the current node (so that the top question of the tree will be asked next). This is the rule that starts the game by instructing: "if you haven't got a question you're about to ask, ask the topmost question in the tree of questions". The detailed syntax of forward chaining rule definitions will be explained in 3.1 Forward chaining.

Select "-- All backward rules --" from the drop-down list and bring up a backward chaining rule definition by clicking on its name in the Rule Browser and choosing Rule > Find Source again. The detailed syntax of backward chaining rules is in 3.2 Backward Chaining.

2.4.2 Objects Browser

KnowledgeWorks Objects Browser

The Objects Browser is for exploring the contents of the KnowledgeWorks object base. Start it by choosing KnowledgeWorks > Objects. The system knows about the CLOS objects that make up the object base. One class of CLOS objects in this example is the node class so choose NODE from the Preset query/pattern drop-down. All the node objects in the object base will be displayed in the pane below. Click on one of these objects and the bottom pane will display the slots.
To make the display clearer and allow input without explicit package qualifiers, change the package of the Objects Browser. Do this via **Works > Tools > Preferences... > Objects Browser > Package**. Edit the Package pane so that it says **KW–USER** and press **OK**.

Now change the **Query** field to read `(node ?object animal ?a)` and press **Return**. The animals associated with each node are displayed. In this game there is a tree of questions with each node object representing a question. Some nodes have a **nil** value for the animal slot; these are the non-terminal nodes in the question tree. The program learns your new animals by adding new nodes to the tree.

Now type `?a` into the **Pattern** field (and press **Return**). This displays only the animals. The values displayed in the topmost of the two panes is the **Pattern** field instantiated with every possible object that matches the **Query** field. However, if the **Pattern** field is empty then the value of the **Query** field is taken to be the pattern.

Change the **Query** field to read `(and (node ?n animal ?a) (test ?a))` and press **Return**.

Objects Browser matching animals

![Objects Browser Matching Animals](image)

Only the non-nil animals are displayed.

### 2.4.3 Class Browser
KnowledgeWorks Class Browser

The Class Browser is obtained by choosing KnowledgeWorks > Classes. This brings up the LispWorks Class Browser with an initial focus on the class standard-kb-object. Select the Subclasses tab to display the subclasses of standard-kb-object. Double click on NODE in the subclasses pane to examine the node class used in this tutorial. Select the Slots tab to display its slots and click on one of the slots in the middle pane, for example the ANIMAL slot. This displays more information about the slot in the Description pane.

Other useful features of the Class Browser include the Superclasses tab which display a graph of the superclasses; the Hierarchy tab which displays direct superclasses and subclasses; and the Functions tab which displays the generic functions or methods defined on a class either directly or through inheritance. For more information about the Class Browser, see the LispWorks IDE User Guide.
2.4.4 Forward Chaining History

KnowledgeWorks Forward Chaining History

This is obtained by choosing KnowledgeWorks > FC History. If you have just run the tutorial a window will appear of which the left column contains the entry DEFAULT-CONTEXT. These are all the contexts (rule groups) the forward chaining engine has executed (in this case only one). On the right is a detailed breakdown of what happened in each cycle within this context. You will see the rule names listed down the left, and the cycle numbers along the top. The boxes indicate which rules fired. In the last cycle, you will see a black box indicating that the rule GAME-FINISHED fired, and a outlined box for the rule PLAY. This means that the rule PLAY could have fired, but that GAME-FINISHED was preferred.

Note: you can remove the package prefixes from displayed symbols by setting the current package of the FC History tool to KW-USER, in the same way as you did for the Objects Browser tool (see 2.4.2 Objects Browser).

Look at the definition for GAME-FINISHED (find the source using the Rule Browser) and notice that it contains :priority 15. This means that the GAME-FINISHED rule has higher priority than the PLAY rule (which has the default value of 10), and so was preferred. Other methods of conflict resolution are also available.

2.5 KnowledgeWorks Listener

The KnowledgeWorks Listener has already been shown to function as a Lisp Listener. However it extends this with the ability of the Objects Browser to match objects. When using the Objects Browser the Query pane contained patterns which could be matched against the Object Base. These same patterns can be entered into the KnowledgeWorks Listener. Enter (node ?object) into the Listener. This asks "Are there any node objects?". A NODE object will be returned. To ask for more solutions press the Next button. If there are more you will be shown another, otherwise the listener displays the word NO and the listener prompt reappears. If you do not want to see any more, just press the Return key.

Try entering some of the other expressions from the Objects Browser, for example (and (node ?n animal ?a) (test ?a)). If the input is not recognized it is treated as Lisp.
2.6 Debugging

2.6.1 Monitoring Forward Chaining Rules

KnowledgeWorks Rule Monitor

One of the problems with forward chaining rules is determining why they are (or are not) being matched. To deal with this KnowledgeWorks has Monitor Windows for forward chaining rules. To bring up a Monitor Window, select the DEFAULT-CONTEXT in the Rule Browser, click on PLAY and choose Rule > Monitor. Alternatively you can use the context menu to raise the Rule Monitor window. A Rule Monitor window appears displaying in its upper pane the conditions of the rule. Both are highlighted meaning they are matched (as single conditions without reference to any variable bindings across conditions) in the object base. If you select one or more of these conditions, the message will change from "Number of instantiations matching selected conditions: <n>" to "No instantiations matching selected conditions" depending on whether objects can be found in the object base to match all the selected conditions at once (this takes account of variables bound across conditions).

By selecting the All Unfired Instantiations button, you can list any unfired instantiations of the rule. In this case there is one unfired instantiation. Selecting this in the lower pane and then choosing Instantiations > Inspect raises an Inspector tool displaying the variable bindings in the instantiation.

You can have any number of monitor windows (though at most one per rule). At times (during rule execution, for example)
the object base may change. Monitor windows can be updated by choosing \textit{Works > Refresh} from the Rule Monitor menu bar, or \textit{Memory > Update Monitor Windows} from the KnowledgeWorks Listener. When you are single-stepping through rules (see below) Monitor windows are updated automatically.

\section*{2.6.2 Single-Stepping Rules}

KnowledgeWorks Gspy Window

Select a rule, say, \texttt{Y-N-QUESTION}, in the Rule Browser and choose \textit{Rule > GSpy} from the menu bar. This brings up a Spy Window for the rule. In it you will see the actions of the rule.

Now enter (\texttt{infer}) in the Listener to run the demo again. Execution will stop when this rule fires. A message in the listener will say that the rule \texttt{Y-N-QUESTION} has been called. Click on the \texttt{Creep} button at the bottom of the Listener to single step through the rule. Watch the highlight move through the Spy Window as you go. If you still have a Monitor Window for the \texttt{PLAY} rule it will be updated automatically as you go.

Click on \texttt{Leap} at the bottom of the Listener and it will "leap" to the end of the rule. When you have finished, close the Spy Window (for example by \textit{Works > Exit > Window}) and press \texttt{Leap} in the Listener window to remove the break point and continue normally.

At any point when rule execution is suspended by this mechanism, the other KnowledgeWorks tools may be used, for example to examine the object base (with the Objects Browser) or see which rules have fired (with the forward chaining history). Spy Windows are available for backward chaining rules as well, and they work in exactly the same way (they are set by selecting the rule in the Rule Browser and choosing \textit{Rule > Gspy}).

\section*{2.6.3 Editing Rule Definitions}
Let us suppose that when the demo finishes we would like it to ask if we want to play again. Find the definition for \texttt{GAME-FINISHED} (using the Rule Browser). One line in the definition is commented out with a ; (semi-colon) at the start. Remove the semi-colon and compile the new definition by choosing \textit{Definitions > Compile} from the editor menu bar. Press \texttt{Space} to return to the editor view. This rule will now ask if the user wants to play again and execution will only stop (the \texttt{(return)} instruction ends execution) if requested. Run the demo to see this happen.

The rule \texttt{FETCH-NEW-ANIMAL} also has a commented-out line (repeat) which will make it repeat its prompt until given an animal it does not already know. Remove the semi-colon at the start of the line in and compile the new definition of the rule. Run the demo again and try giving the system an animal it recognizes. It will prompt again. Give it an animal it does not recognize to finish.

\section*{2.7 Lisp Integration}

You can save your object base of animals by entering:

\begin{verbatim}
(save-animals "my-animal-objs.lisp")
\end{verbatim}

into the Listener. In the file of rules "\texttt{animal-rules.lisp}" look at the function \texttt{save-animals} which does this. Note how the Lisp code directly uses the same objects as the rules. If we used the Lisp code to modify the slots of the objects the KnowledgeWorks rule interpreter would keep track.
2.7.1 The LispWorks IDE

The entire programming environment of the LispWorks IDE is available from the menus on the KnowledgeWorks Podium window. See the *LispWorks IDE User Guide* for more details.

2.8 Systems

KnowledgeWorks System Browser

If you are familiar with LispWorks system definitions, look at the system definition for the animal demo, by evaluating:

```lisp
(example-edit-file "kw/animal/defsystem")
```

It contains systems with type `:kb-system` and `:kb-init-system`. Examine the components of each system (which can be source files or subsystems) using the System Browser which is available from the Editor via `Meta+X Describe System` or `File > Browse Parent System`.

Systems with type `:kb-system` are reloaded when the rules are cleared. Systems with type `:kb-init-system` are reloaded when the object base is cleared.

Try this out by finding the KnowledgeWorks Listener and choosing `Memory > Clear Objects and Rules`. Then enter
(load-system "ANIMAL") into the KnowledgeWorks Listener to reload the system animal. Both the files animal-rules and animal-objs are reloaded. Now choose Memory > Clear Objects and reload the animal system again and note how only the file animal-objs is reloaded.

2.9 Exiting KnowledgeWorks

KnowledgeWorks is integrated with LispWorks so you cannot exit from KnowledgeWorks independently. You can close individual KnowledgeWorks windows by choosing Works > Exit > Window. You can exit LispWorks by choosing Works > Exit > LispWorks... from any KnowledgeWorks or LispWorks window. If you have any unsaved edited files you will be asked whether you wish to save them. There will be a final confirmation before KnowledgeWorks quits.
3 Rules

KnowledgeWorks rules are defined as follows:

```
<rule> ::= (defrule <rule-name> <direction> [<doc-string>] <body>)
<direction> ::= {:forward | :backward}
```

Every rule must have a unique name which must also be distinct from any KnowledgeWorks object class name and from any context (rule-group) name. The expressions which form the body of a rule have the same syntax and meaning regardless of whether they occur on the left or right hand side of a forward or backward chaining rule. If doc-string is given, then it should be a string. The value can be retrieved by calling the function `documentation` with doc-type `rule`.

3.1 Forward chaining

3.1.1 Overview

Forward chaining rules consist of a condition part and an action part. The condition part contains conditions which are matched against the object base. If and only if all the conditions are matched, the rule may fire. If the rule is selected to fire, the actions it performs are given in the action part of the rule. The process of selecting and firing a rule is known as the Forward Chaining Cycle, and the forward chaining engine cycles repeatedly until it runs out of rules or a rule instructs it to stop. KnowledgeWorks forward chaining rules reside in a group of rules, or context, and may have a priority number associated with them for conflict resolution (choosing which of a set of eligible rules may fire).

3.1.2 Forward Chaining Syntax

Forward chaining rule bodies are defined by:

```
<body> ::= [:context <context-name>] [:priority <priority-number>] <forward-condition>* --> <expression>*
```

where `<context-name>` is the name of a context which has already been defined (see 3.1.5 Control Flow) defaulting to `default-context`, and `<priority-number>` is a number (see 3.1.5 Control Flow) defaulting to 10.

The syntax for forward-conditions is:

```
<forward-condition> ::= <object-condition> |
| (test <lisp-expr>) |
| (not <forward-condition>+) |
| (logical <forward-condition>+)

<object-condition> ::= (<class-name> <variable> [object-slot-condition]*)
```
3 Rules

\[
<\text{object-slot-condition}> ::=
  <\text{slot-name}> <\text{term}>
\]

\(<\text{object-condition}>\) is an object-base match where the variables (introduced by "?" ) in <\text{term}> are bound (via destructuring) to the corresponding data in the slot named by <\text{slot-name}>. <\text{variable}> is a single variable bound to the object matched.

Note: "?" on its own denotes an anonymous variable which always matches.

\(<\text{test} <\text{lisp-expr}>\) is a Lisp test where <\text{lisp-expr}> is any Lisp expression using the variables bound by other conditions, and which must succeed (return non-nil) for the condition to match. Computationally cheap Lisp tests can frequently be used to reduce the search space created by the object base conditions. Lisp tests, and any functions invoked by them, should not depend on any dynamic global data structures, as changing such structures (and hence the instantiations of the rule) will be invisible to the inference engine. Lisp tests can depend on the values of slots in objects matched by preceding object-base conditions only if the values are bound to variables in the rule using the <\text{object-slot-condition}> syntax. They cannot depend on values obtained by calling \text{slot-value} or a reader function.

\(<\text{not} <\text{forward-condition}>+\) is simply a negated condition. A negated condition never binds any variables outside its scope. Variables not bound before the negation will remain unbound after it.

\(<\text{logical} <\text{forward-condition}>+\) is used to indicate clauses that describe the \textit{logical dependencies} amongst objects. See 6.4 Logical Dependencies and Truth Maintenance for more details.

Note that if a forward chaining rule contains any conditions at all then it must contain at least one object base reference of the form:

\[
(<\text{class-name} <\text{variable}> ...)
\]

The syntax for expressions is:

\[
<\text{expression}> ::=
  <\text{forward-condition}>
  |(\text{erase} <\text{variable}>)
  |(\text{assert} (<\text{class-name} <\text{variable}>
  [<\text{slot-name} <\text{term}>]*))
  |(\text{context} <\text{context-list}>)
  |(\text{return})
  |(<\text{lisp-expr} <\text{term}>*)
  |<\text{goal}>
\]

<\text{forward-condition}> is a forward condition which must succeed for execution of the action part of the rule to continue.

\(<\text{erase} <\text{variable}>\) removes the instance bound to <\text{variable}> from the knowledge base. It is an error if <\text{variable}> is bound to anything but a KnowledgeWorks instance.

\(<\text{assert} (<\text{class-name} <\text{variable}>
  [<\text{slot-name} <\text{term}>]*))\) is an assertion which modifies the contents of the object base, where if <\text{variable}> is unbound a new object of the given class with the given slot-values is created, and if it is bound, the object to which it is bound has its slots modified to the given values.

\(<\text{context} <\text{context-list}>\) adds the given list of contexts to the top of agenda (see 3.1.5 Control Flow).

\(<\text{return}>\) passes control to the top context on the agenda and removes it from the agenda (see 3.1.5 Control Flow).

\(<\text{lisp-expr} <\text{term}>*)\) binds the result or results of calling \text{lisp-expr} to the <\text{term}>s with execution of the rule terminating if any bindings fail (if no <\text{term}>s are given execution will always continue).

<\text{goal}> may be any backward chaining goal expression (see 3.2 Backward Chaining).

Note that in the action part of a rule, only backward chaining goals and object base matches invoke the backward chainer.
3 Rules

3.1.2.1 Example

(defun move-train :forward
  :context train
  (train ?train position ?train-pos)
  (signal ?signal position ?signal-pos color green)
  (test (= ?signal-pos (1+ ?train-pos)))
  -->
  ((format t "Train moving to position ~s\n" ?signal-pos))
  (assert (signal ?signal color red))
  (assert (train ?train position ?signal-pos)))

specifies that if there is a train with a green signal directly in front then the train may move on and the signal changes to red.

3.1.3 Defining Forward Chaining Rules

Forward chaining rules may be defined and redefined incrementally. When redefined all the instantiations of the rule are recreated. This means that during execution of a rulebase the redefinition capability should be used with care as previously fired instantiations will reappear and may fire again.

When a rule is redefined it inherits its order (with respect to the order conflict resolution tactic) from its initial definition. If this is not required, the rule should be explicitly undefined before being redefined.

A forward chaining rule may be undefined by entering:

(undefrule <rule-name>)

A warning will be given if the rule does not exist.

3.1.3.1 Example

(undefrule move-train)

3.1.4 The Forward Chaining Interpreter

The forward chaining rule interpreter may be invoked by the Lisp function:

(infer [:contexts <context-list>])

where <context-list> is a list of contexts where control is passed immediately to the first in the list, and the rest are placed at the top of the agenda. The object base may or may not be empty when the forward chainer is started. The infer function returns the final cycle number. When not specified, <context-list> defaults to (default-context).

3.1.5 Control Flow

3.1.5.1 The Agenda

The agenda is essentially a stack of rule groups (called contexts) which are still awaiting execution. The initial invocation of the forward chainer and any subsequent rule can cause contexts to be added to the top of the agenda. During normal execution the forward chainer simply proceeds down the agenda context by context. When the agenda is empty, passing control on will terminate the execution of the rule interpreter. This is a proper way to exit the forward chainer.


3.1.5.2 Contexts

Contexts are the groups into which rules are partitioned. The context `default-context` always exists. Contexts are defined by:

```lisp
<context> ::= (defcontext <context-name>
  [:strategy <CRS>]
  [:auto-return t | nil]
  [:meta <meta-actions>]
  [:documentation <doc-string>])
```

where `<context-name>` is a symbol, `<CRS>` is a conflict resolution strategy defaulting to `(priority recency order)` (see below). If `:auto-return` is set to `t` (the default) then when the context has no more rules to fire, control passes to the next context on the agenda, but if it is `nil` an error occurs (a rule in the context should have issued a `(return)` instruction explicitly). The `:meta` option is necessary only if the default behavior of the context is to be modified and is explained in 6.1.1 Meta Rule Protocol. If `:documentation` is given, then `doc-string` should be a string and the value can be retrieved by calling the function `documentation` with doc-type `context`.

3.1.5.3 Conflict Resolution

Every context has its own conflict resolution strategy, specified in the `defcontext` form. A conflict resolution strategy is an ordered list of conflict resolution tactics. A conflict resolution tactic may be any of the following:

- **priority**
  Instantiations of rules with the highest priority are preferred.

- **-priority**
  Instantiations of rules with the lowest priority are preferred.

- **recency**
  The most recently created instantiations are preferred.

- **-recency**
  The least recently created instantiations are preferred.

- **order**
  Instantiations of rules defined/loaded earliest are preferred. This favors the topmost rules in a file.

- **-order**
  Instantiations of rules defined/loaded latest are preferred.

- **specificity**
  The most specific rules are preferred (specificity is a score where a point is awarded for every occurrence of a variable after the first, every Lisp test, and every destructuring expression; the highest score wins).

- **-specificity**
  The least specific rules are preferred.

- **mea**
  MEA stands for Means End Analysis. Instantiations are preferred where the object corresponding to the topmost object-matching condition is more recently modified.

- **-mea**
  Instantiations are preferred where the object corresponding to the topmost object-matching condition is less recently modified.

- **lex**
  LEX stands for LEXicographic. Each instantiation is represented by the (in descending order) sorted list of the most recently modified cycle numbers of the objects in the instantiation; these lists are compared place by place with an instantiation being preferred if it first has a larger number in a particular position, or if it runs out first (hence the analogy with lexicographic ordering).

- **-lex**
  The converse of the above.

The tactics are applied successively starting with the left-most until only one instantiation is left or until all tactics have been applied when it is unspecified which of the resulting set is chosen. For example, using the strategy `(priority recency)`
first all the instantiations which are not of the highest priority rule or rules (as given by the rule's priority number) are discarded and then all instantiations which were not created in the same forward chaining cycle as the most recently created instantiation will be discarded. If more than one instantiation is left it is unspecified which will be selected to fire.

Note that the strategy (lex specificity) is equivalent to the OPS5 strategy LEX and (mea lex specificity) is equivalent to the OPS5 strategy MEA, hence the borrowing of these terms. For further information on LEX and MEA in OPS5 the reader is referred to Programming Expert Systems in OPS5, by Brownston, Farrell, Kant and Martin (published by Addison-Wesley). However, KnowledgeWorks is not heavily optimized to use the tactics mea, -mea, lex or -lex.

3.1.6 Examples

(defcontext trains
  :strategy (priority recency order)
  :auto-return t)
(defcontext trains)

These two definitions are in fact equivalent.

3.1.6.1 Defining Contexts

A context may be defined and redefined. Redefining a context will clear all the rules in the context.

A context may be undefined and removed by entering:

(undefcontext <context-name>)

3.1.7 Forward Chaining Debugging

Forward chaining debugging may be turned on by typing:

(all-debug)

and off by typing:

(no-debug)

When KnowledgeWorks is started, debugging is on. Debugging allows the actions of forward chaining rules to be single-stepped like backward chaining rules (see 3.2.7 Backward Chaining Debugging), and also records information on which objects are modified by which rules. For information on how to use the debugging tools, refer to 5 The Programming Environment.

3.2 Backward Chaining

3.2.1 Overview

Backward chaining involves trying to prove a given goal by using rules to generate sub-goals and recursively trying to satisfy those. The KnowledgeWorks backward chaining engine is an extension of the LispWorks Common Prolog system which can match directly over KnowledgeWorks CLOS objects (the object base). All the standard Common Prolog facilities and built-in predicates are available. For more detailed information the reader is referred to the Appendix A: Common Prolog. Note that all the different ways of proving a particular goal are defined together in the same form.
3 Rules

3.2.2 Backward Chaining Syntax

Backward chaining rule bodies are defined as:

\[
\begin{align*}
<\text{body}> & ::= <\text{clause}>+ \\
<\text{clause}> & ::= ( <\text{goal}> \leftarrow <\text{expression}> )* \\
<\text{goal}> & ::= ( <\text{rule-name}> <\text{term}> )* \\
\end{align*}
\]

In each sub-clause of the rule, the goal must have the same arity (number of arguments). Within each \textit{<term>} destructuring is allowed and variables are introduced by \(?\) (and \(?\) on its own denotes the anonymous variable which always matches).
\textit{<expression>} is as defined in 3.1.2 Forward Chaining Syntax.

3.2.2.1 Example

\begin{verbatim}
(defrule link-exists :backward
  ((link-exists ?town1 ?town2)
   \leftarrow
   (or (link ?link town1 ?town1 town2 ?town2)
       (link ?link town2 ?town1 town1 ?town2))
   (cut))
((link-exists ?town1 ?town2)
 \leftarrow
 (route-exists ?town1 ?town2)))
\end{verbatim}

which says that a link exists between two towns either if there is a link object between them in the object base or if there is a route between the towns. The \textit{route-exists} predicate would be defined by another backward chaining rule, or might be in the Prolog database.

3.2.3 Objects

Backward chaining rules may refer to the object base using the standard \((<\text{class-name}> <\text{variable}> [<\text{slot-name}> <\text{term}>]*)\) syntax, and these expressions are instantiated directly without creating any sub-goals. The \textit{<class-name>} of any CLOS class or KnowledgeWorks structure may not coincide with any backward chaining \textit{<rule-name>}. The Common Prolog database may be used to record factual information but it is distinct from the object base in that it may contain variables, and anything in it is inaccessible to the forward chaining rule preconditions.

3.2.4 Defining Backward Chaining Rules

Backward chaining rules may be defined and redefined incrementally.

3.2.5 The Backward Chaining Interpreter

The backward chaining interpreter can be invoked from Lisp by the following functions:

\[
\begin{align*}
\text{(any expr-to-instantiate expr-to-prove)} \\
\text{(findall expr-to-instantiate expr-to-prove)}
\end{align*}
\]

which finds any solution to \textit{expr-to-prove} and instantiates \textit{expr-to-instantiate}, and:

\[
\begin{align*}
\text{(findall expr-to-instantiate expr-to-prove)}
\end{align*}
\]

finds all the solutions to \textit{expr-to-prove}, instantiates \textit{expr-to-instantiate} for each and returns these in a list.

For other interface functions to be called from Lisp the reader is referred to Appendix A: Common Prolog.
3 Rules

From the action part of a forward chaining rule the backward chainer is called implicitly when a CLOS match or goal expression is used. The action part of forward chaining rules and the antecedents of backward chaining rules are syntactically and semantically identical.

3.2.5.1 Examples

(\(\text{any \ '}(?x \ \text{is \ in \ (1 \ 2 \ 3)) \ '}(\text{member \ ?x \ (1 \ 2 \ 3)))\))

returns:

(1 \ \text{is \ in \ (1 \ 2 \ 3))}

The following expression:

(findall \ '(?x \ \text{is \ in \ (1 \ 2 \ 3)) \ '}(\text{member \ ?x \ (1 \ 2 \ 3)))

returns:

((1 \ \text{is \ in \ (1 \ 2 \ 3)))) (2 \ \text{is \ in \ (1 \ 2 \ 3))) (3 \ \text{is \ in \ (1 \ 2 \ 3)))

3.2.6 Edinburgh Prolog Translator

Edinburgh syntax Prolog files may be compiled and loaded if they are given .pl as a file extension. These are completely compatible with the KnowledgeWorks backward chaining rules. For more details refer to A.10 Edinburgh Syntax.

3.2.7 Backward Chaining Debugging

Backward chaining debugging follows the Prolog four port model. Backward chaining rules may be ”spied” (this is a Prolog term which corresponds to tracing and single-stepping) which puts a break-point on them and means they can be single-stepped when they are invoked. When forward chaining debugging is on, the action part of forward chaining rules can be spied and single-stepped in the same way when they are fired. 5 The Programming Environment, explains this in detail. The leashing of the ports can be adjusted, details are to be found in A.7 Debugging.

3.3 Common Lisp Interface

Arbitrary Lisp expressions may be called from rules. See 3.1.2 Forward Chaining Syntax.
4 Objects

The object base contains KnowledgeWorks CLOS objects (including relational database objects) and KnowledgeWorks structures. KnowledgeWorks CLOS objects can be treated as ordinary CLOS objects and may be manipulated directly from Lisp. KnowledgeWorks relational database objects may transparently retrieve their slot values from a relational database using the LispWorks object-oriented relational database interface.

KnowledgeWorks structures are more efficient but reduced functionality CLOS objects similar in spirit to Lisp structures. Values in the slots of these objects should not be destructively modified unless these values are themselves KnowledgeWorks objects. This is because the rule interpreter keeps track of the changes to the slots, and a destructive operation is likely to bypass this process.

4.1 CLOS objects

A KnowledgeWorks CLOS class may not have a class name which coincides with any rule, context or KnowledgeWorks structure (See 4.3 KnowledgeWorks Structures). KnowledgeWorks CLOS classes fall into one of two categories, either unnamed or named. Named objects can be given a name (or they use a default name) and can be referred to by name. Otherwise, named and unnamed objects have equivalent functionality. CLOS objects may be made by the Common Lisp function make-instance, taking the same arguments. An unbound slot will return :unbound until set.

Name clashes are arbitrated by *signal-kb-name-clash* and signal an error by default. See the reference manual page.

4.1.1 Unnamed Classes

Unnamed classes may be defined by the macro def-kb-class which takes the same arguments as the defclass macro. It is identical to using defclass and supplying the KnowledgeWorks mixin standard-kb-object if none of the superclasses already contains it. The function make-instance may be used to create instances of the class.

4.1.2 Named Classes

A named KnowledgeWorks CLOS class is defined by the macro def-named-kb-class which is syntactically identical to the Common Lisp defclass macro, and semantically identical with the exception that it adds a KnowledgeWorks mixin class named-kb-object if none of the superclasses already contains it, and makes the default name for the objects be a symbol generated from the class name. Classes defined by def-named-kb-class contain a name slot which those defined by def-kb-class do not.

The function make-instance can be given the initialization argument :kb-name to specify a name. If not specified, a default name is generated from the name of the class. All names must be distinct as regarded by eq. The function:

```lisp
(get-kb-object <name>)
```

retrieves the instance from its name. The function:

```lisp
(kb-name <object>)
```

returns the name of the given object.
4.1.2.1 Examples

```
(def-named-kb-class truck ()
  ((location :initarg :location)
   (destination :initarg :destination)))
(make-instance 'truck
  :kb-name 'ford1
  :location 'Cambridge)
```

creates the instance `<KB-OBJECT FORD1>`.

```
(make-instance 'truck :location 'London)
```

creates the instance `<KB-OBJECT TRUCK123>`, and:

```
(get-kb-object 'ford1)
```

returns `<KB-OBJECT FORD1>` and:

```
(kb-name (get-kb-object 'ford1))
```

returns FORD1. The class definition:

```
(defclass truck (named-kb-object) ...)
```

would have been identical except that the second truck would have been given a name such as `OBJECT345` rather than `TRUCK123` (as `def-named-kb-class` overrides the inherited initform for the `kb-name` slot `(gentemp "OBJECT")` with a more specific one `(gentemp <class-name>)`).

4.2 Relational Database Objects

A CLOS/SQL class may also be given the KnowledgeWorks mixin class, enabling rules to refer to these objects as if there were no database present. However, their database functionality carries over transparently. For example, consider the case where a slot in the database class is designated for deferred retrieval from the database. When the rulebase queries the contents of the slot, a database query will automatically be generated to retrieve and fill in the value of the slot, and the rulebase will continue as if the value had been there in the first place.

Details of the LispWorks Common SQL interface can be found in the *LispWorks® User Guide and Reference Manual*.

4.2.1 Example

```
(sql:def-view-class vehicle
  (standard-db-object standard-kb-object)
  ((vehicle_no :db-kind :key)
   (keeper)
   (owner :db-kind :join
     :db-info (:home-key :keeper
       :foreign-key person_id
       :retrieval :deferred
       :join-class person))))
```

defines a database class `vehicle` where the `person` object in the `keeper` slot is retrieved from the `person` table in the database using the value of the `keeper` slot as key, only when queried. In the list of superclasses, `standard-kb-object` should appear after `sql:standard-db-object`. 
4.2.2 Extended Example

The following example is a complete segment of code which allocates person objects to vehicle objects. Note how once the class definitions have been made, the rules do not in any way reflect the fact that there is an underlying database. The example output assumes a database initialized by the following SQL statements:

```sql
drop table VEHICLE;
cREATE TABLE VEHICLE
(PLATE CHAR(8) NOT NULL, MAKE CHAR(20),
PRICE INTEGER, OWNER CHAR(20));
grant all on VEHICLE to public;
insert into VEHICLE values
('E265 FOO', 'VAUXHALL', 5000, '');
insert into VEHICLE values
('XDG 792S', 'ROLLS', 50000, '');
insert into VEHICLE values
('F360 OOL', 'FORD', 4000, 'PERSEPHONE');
insert into VEHICLE values
('H151 EEE', 'JAGUAR', 15000, '');
insert into VEHICLE values
('G722 HAD', 'SKODA', 500, '');
drop table PERSON;
cREATE TABLE PERSON
(NAME CHAR(20) NOT NULL, SALARY INTEGER, VEHICLE CHAR(8),
EMPLOYER CHAR(20));
insert into PERSON values ('FRED', 10000, '', 'IBM');
insert into PERSON values ('HARRY', 20000, '', 'FORD');
insert into PERSON values ('PHOEBE', 5000, '', '');
insert into PERSON values ('TOM', 50000, '', 'ACME');
insert into PERSON values ('PERSEPHONE', 15000, 'F360 OOL', 'ICL');
drop table COMPANY;
cREATE TABLE COMPANY
(NAME CHAR(20), PRODUCT CHAR(10));
insert into COMPANY values ('IBM', 'COMPUTERS');
insert into COMPANY values ('FORD', 'CARS');
insert into COMPANY values ('ICL', 'COMPUTERS');
insert into COMPANY values ('ACME', 'TEAPOTS');
```

Below is an example rulebase that analyzes the database and outputs a suggestion as to which vehicle should be allocated to which person. The full code and the SQL statements to set up the database are included in the examples distributed with KnowledgeWorks.

```
(in-package "KW-USER")

;;; the vehicle class maps onto the car table in the
database owner is a join slot which looks up the
owner person object

(sql:def-view-class vehicle
 (sql:standard-db-object standard-kb-object)
 ((number-plate :accessor vehicle-number-plate
 :type (string 8)
 :db-kind :key
 :column plate)
 (make :accessor vehicle-make
 :type (string 20)
 :db-kind :base
 :column make)
 (price :accessor vehicle-price
 :type integer
 :db-kind :base)
```

32
(def-view-class person
  (standard-db-object standard-kb-object)
  ((name :accessor person-name
        :type (string 20)
        :db-kind :key
        :column name)
   (salary :accessor person-salary
           :type integer
           :db-kind :base
           :column salary)
   (vehicle-number-plate :type (string 8)
                         :db-kind :base
                         :column vehicle)
   (vehicle :accessor person-vehicle
            :db-kind :join
            :db-info (:home-key vehicle-number-plate
                      :foreign-key number-plate
                      :join-class vehicle
                      :set nil
                      :retrieval :deferred)))

(def-view-class company
  (standard-db-object standard-kb-object)
  ((name :accessor company-name
          :type (string 20)
          :db-kind :key
          :column name)
   (product :accessor company-product
            :type (string 10)
            :db-kind :base
            :column product)))

;;; here we assume we have a database connected with
;;; the correct data in it - if we do we retrieve all
;;; the person and vehicle objects but company objects
;;; will be retrieved only when needed by querying
;;; the company slot of the person objects

(if sql:*default-database*
  (progn (sql:select 'vehicle)
         (sql:select 'person))
  (format t
    "Please connect to a database with
    contents ~ created by file data.sql")

;;; to store which vehicles a person can drive
(def-kb-struct vehicles-for-person person vehicles)
(defcontext database-example :strategy (priority))

;;; for every person initialize the list of vehicles they
;;; can drive

(defrule init-vehicles-for-person :forward
  :context database-example
  (person ?person vehicle nil)
  =>
  (assert (vehicles-for-person ?person ?person vehicles nil)))

;;; for every vehicle a person can drive which has not yet
;;; been included in the list, add it to the list

(defrule vehicle-for-person :forward
  :context database-example
  (person ?person vehicle nil)
  (vehicle ?vehicle owner nil)
  (test (not (member ?vehicle ?vehicles)))
  =>
  (vehicle-ok-for-person ?vehicle ?person)
  ; check if ok to drive vehicle
  (assert (vehicles-for-person ?c-f-p vehicles
                               (?vehicle . ?vehicles))))

;;; rules expressing what vehicles a person can drive:
;;; if they have no employer they can only drive a
;;; Skoda otherwise they will refuse to drive a Skoda.
;;; anyone will drive a Rolls or a Jaguar.
;;; they'll only drive a Ford or Vauxhall if salary is
;;; less than 40k.

(defrule vehicle-ok-for-person :backward
  ((vehicle-ok-for-person ?vehicle ?person)
   <--
   (person ?person company nil)
   (cut)
   (vehicle ?vehicle make "SKODA")
   (vehicle-ok-for-person ?vehicle ?person)
   <--
   (vehicle ?vehicle make "SKODA")
   (cut)
   (fail))
  ((vehicle-ok-for-person ?vehicle ?person)
   <--
   (or (vehicle ?vehicle make "ROLLS")
       (vehicle ?vehicle make "JAGUAR")
   (cut))
  ((vehicle-ok-for-person ?vehicle ?person)
   <--
(or (vehicle ?vehicle make "VAUXHALL")
    (vehicle ?vehicle make "FORD")
    (person ?person salary ?salary)
    (test (< ?salary 40000)))

;;; next to rules are just simple allocation rules,
;;; trying out each possibility until one fits

(defrule alloc-vehicles-to-persons :backward
  ((alloc-vehicles-to-persons ?allocs)
   <--
   (alloc-internal nil nil nil ?allocs)))

(defrule alloc-internal :backward
  ((alloc-internal ?done-persons ?done-vehicles
    ?allocs ?allocs)
   <--
   (not (and (vehicles-for-person ? person ?person)
             (not (member ?person ?done-persons)))))
  (cut)

  ((alloc-internal ?done-persons ?done-vehicles
    ?allocs-so-far ?allocs)
   <--
   (vehicles-for-person ? person ?person
    vehicles ?vehicles)
   (not (member ?person ?done-persons))
   (member ?vehicle ?vehicles)
   (not (member ?vehicle ?done-vehicles))
   (alloc-internal (?person . ?done-persons)
     (?vehicle . ?done-vehicles)
     (?allocs-so-far . ?allocs))
   (alloc-internal (?person . ?done-persons)
     (?vehicle . ?done-vehicles)
     (?allocs-so-far . ?allocs)))

;;; find a solution and print it out

(defrule find-solution :forward
  :context database-example
  :priority 5
  (not (not (vehicles-for-person ?))))
  -->
  (alloc-vehicles-to-persons ?solution)
  (dolist (pair ?solution)
    (format t "~%~A drives ~A"
            (person-name (car pair))
            (vehicle-number-plate (cdr pair))))))

Below is sample output from the rulebase with SQL recording turned on to demonstrate the SQL statements that are automatically passed to the database by manipulating the objects:

KW-USER 53 > (infer :contexts '(database-example))
(SELECT VEHICLE.PLATE,VEHICLE.MAKE,VEHICLE.PRICE,VEHICLE.OWNER FROM VEHICLE WHERE (VEHICLE.PLATE = 'F360 OOL'))
(SELECT VEHICLE.PLATE,VEHICLE.MAKE,VEHICLE.PRICE,VEHICLE.OWNER FROM VEHICLE WHERE (VEHICLE.PLATE = ''))
(SELECT VEHICLE.PLATE,VEHICLE.MAKE,VEHICLE.PRICE,VEHICLE.OWNER FROM VEHICLE WHERE (VEHICLE.PLATE = ''))
(SELECT PERSON.NAME,PERSON.SALARY,PERSON.VEHICLE,PERSON.EMPLOYER FROM PERSON WHERE (PERSON.NAME = ''))
(SELECT VEHICLE.PLATE,VEHICLE.MAKE,VEHICLE.PRICE,VEHICLE.OWNER FROM VEHICLE WHERE (VEHICLE.PLATE = ''))
(SELECT PERSON.NAME,PERSON.SALARY,PERSON.VEHICLE,PERSON.EMPLOYER FROM PERSON WHERE (PERSON.NAME = ''))
(SELECT VEHICLE.PLATE,VEHICLE.MAKE,VEHICLE.PRICE,VEHICLE.OWNER FROM VEHICLE WHERE (VEHICLE.PLATE = ''))

35
WHERE (VEHICLE.PLATE = '')
(SELECT
    PERSON.NAME, PERSON.SALARY, PERSON.VEHICLE, PERSON.EMPLOYER
FROM PERSON WHERE (PERSON.NAME = ''))
(SELECT
    PERSON.NAME, PERSON.SALARY, PERSON.VEHICLE, PERSON.EMPLOYER
FROM PERSON WHERE (PERSON.NAME = '')
(SELECT
    PERSON.NAME, PERSON.SALARY, PERSON.VEHICLE, PERSON.EMPLOYER
FROM PERSON WHERE (PERSON.NAME = 'PERSEPHONE'))
(SELECT COMPANY.NAME, COMPANY.PRODUCT FROM COMPANY
WHERE (COMPANY.NAME = 'FORD'))
(SELECT COMPANY.NAME, COMPANY.PRODUCT FROM COMPANY
WHERE (COMPANY.NAME = 'ACME'))
(SELECT COMPANY.NAME, COMPANY.PRODUCT FROM COMPANY
WHERE (COMPANY.NAME = 'IBM'))
(SELECT COMPANY.NAME, COMPANY.PRODUCT FROM COMPANY
WHERE (COMPANY.NAME = ''))
HARRY drives E265 FOO
TOM drives XDG 792S
FRED drives H151 EEE
PHOEBE drives G722 HAD

4.3 KnowledgeWorks Structures

An optimization for improved performance is to replace CLOS objects by KnowledgeWorks structures when the objects are not needed outside the rules, or the full power of object-oriented programming is not required. Within rules they behave the same, although they are not proper CLOS objects. This is discussed in detail in 6.2 Optimization.
5 The Programming Environment

The KnowledgeWorks programming environment is designed for the development of rules. KnowledgeWorks applications will typically contain a mixture of programming styles and so the LispWorks programming environment is available from the menus on the KnowledgeWorks Podium. This chapter deals with KnowledgeWorks specific tools but see the LispWorks IDE User Guide for more details on the LispWorks tools.

KnowledgeWorks Podium

All KnowledgeWorks windows except the Podium can be closed independently of the others by choosing Works > Exit > Window. You can switch between windows by choosing Windows > window-name. The contents of the KnowledgeWorks menu on the KnowledgeWorks Podium is also available by choosing Works > KnowledgeWorks on each tool.

5.1 The KnowledgeWorks Listener

The KnowledgeWorks Listener is obtained by choosing KnowledgeWorks > Listener. This tool is based on the LispWorks Common Prolog Logic Listener (see Appendix A: Common Prolog for further details). Input is taken as being a goal expression to be satisfied unless no predicate of that name and arity (number of arguments) exists in which case it is taken as a Lisp expression. That is, the input may be either:
as defined in 3.1 Forward chaining, or:

with the former interpretation taking priority when ambiguous. Interaction is Prolog-style, so when the bindings which satisfy a goal are printed, pressing Return terminates execution, and entering ; (semi-colon) and Return (or just clicking on the Next button at the bottom) looks for the next solution to the goal.

The File, Leashing and Spy menus behave as for the Common Prolog Logic Listener (see Appendix A: Common Prolog) and the Values, Debug and History menus behave as for the Lisp Listener (see the LispWorks IDE User Guide).

5.2 The Editor

KnowledgeWorks Editor

The KnowledgeWorks Editor is created by choosing KnowledgeWorks > Editor. It is the same as the LispWorks Editor tool. Please see the LispWorks IDE User Guide for more information on the editor tool and the Editor User Guide for information on the various editing commands.
5.3 Clearing KnowledgeWorks

The KnowledgeWorks object base (all the KnowledgeWorks CLOS objects and any optimized structures) may be cleared by choosing Memory > Clear Objects from the KnowledgeWorks Listener, or by calling the function reset.

KnowledgeWorks rules may be cleared by choosing Memory > Clear Rules from the KnowledgeWorks Listener, or by calling the function clear-rules. Clearing the rules does not remove the default context default-context but all the rules in it are removed.

KnowledgeWorks object base and rules may be cleared by choosing Memory > Clear Objects and Rules from the KnowledgeWorks Listener, or by calling the function clear-all. CLOS class definitions remain in effect.

5.4 The System Browser

KnowledgeWorks System Browser

The KnowledgeWorks system browser is obtained by choosing KnowledgeWorks > Systems. It is the same as the LispWorks System Browser, but includes new types of system:

- :kb-system, which are reloaded when the KnowledgeWorks rules are cleared (see 5.3 Clearing KnowledgeWorks).
- :kb-init-system, which are reloaded when the KnowledgeWorks object base is cleared (see 5.3 Clearing KnowledgeWorks).
For more information on LispWorks systems, see the Common Defsystem chapter in the \textit{LispWorks\textsuperscript{®} User Guide and Reference Manual}. For more information about the System Browser tool, see the \textit{LispWorks IDE User Guide}.

## 5.5 The Class Browser

KnowledgeWorks Class Browser

The Class Browser is obtained by choosing \textbf{KnowledgeWorks > Classes}. It is the same as the LispWorks Class Browser except that:

- It appears with an initial focus on \texttt{standard-kb-object}.
- When looking at a KnowledgeWorks class the \textbf{Classes} menu and context menu contain an \texttt{Inspect Instances} command which allows you to look at the instances of the class.
Inspecting instances from the Class Browser

This raises an Inspector tool with a list of all the instances.
Any of the instances displayed in the lower pane may itself be inspected by double-clicking on it.

Other options available in the Class Browser include:

- **Superclasses** and **Subclasses** tabs to draw a graphs of the superclasses or subclasses of the class being looked at.
- **Slots** and **Initargs** tabs to show how the instances can be accessed and initialized.
- **Functions** tab to show the generic functions or methods defined on this class, either directly or by inheritance.

Additionally the **Classes** menu contains a **Browse Metaclass** command which browses the class of this class.

Further details can be found in the *LispWorks IDE User Guide*.

**5.6 The Objects Browser**
KnowledgeWorks Objects Browser

The Objects Browser is obtained by choosing KnowledgeWorks > Objects. Any <expression> (See 3.1 Forward chaining) may be entered into the Query pane. This expression may be a query about the object base or any expression for the backward chainer to prove. The Pattern pane contains the pattern to be instantiated for each solution of the query. If left blank, the pattern used is the query itself.

The Show Inferencing State dropdown allows you to choose which named inferencing state is used to supply the object base for the query.

The Preset query/pattern pane offers a convenient way to examine instances on a per-class basis. All the instances of a class class-name known to KnowledgeWorks (either a CLOS class or a KnowledgeWorks structure class) may be examined by selecting class-name, and all the instances in the object base may be viewed by selecting All classes.

The package used to read and print symbols may be modified by choosing Works > Tools > Preferences... > Objects Browser > Package and entering a package name into the Package pane. Clicking OK will update the tool.

The pane below the query displays all the instantiations of the query, and if the entries refer to an object (so are of the form (<class-name> <object> ...) or just <object>) double-clicking on them will display the slot names and values, and information on when the object was created or modified (if debugging is turned on) in the bottom pane. The selected query
item may be inspected by choosing **Instantiations > Inspect**.

The Objects Browser may be updated by positioning the mouse in either the **Query** or the **Pattern** pane and pressing **Return** or by choosing **Works > Refresh**.

### 5.7 The Rule Browser

KnowledgeWorks Rule Browser

The Rule Browser may be obtained by choosing **KnowledgeWorks > Rules**. It displays contexts and their rules. The **Contexts** pane at the top allows you to select from a drop-down list either a forward chaining context or the special pseudo-context containing all the backward chaining rules. The **Rules** pane lists the rules for the selected context.

The **Context** menu acts on the selected context. Choosing **Context > Find Source** will bring up the definition of the context in the file where it was defined, and choosing **Context > Gspy** will bring up a Spy Window (see **5.8 Debugging with the Environment**) for the context, displaying the meta-interpreter (see **6.1.1 Meta Rule Protocol**) for the context if one is defined. If debugging is turned on a meta-interpreter is always defined. Choosing **Context > NoGspy** will remove the Spy Window (see **5.8 Debugging with the Environment**).

The **Rule** menu acts on the rule selected in the lower pane. All rules may be edited by choosing **Rule > Find Source**. Spy Windows can be brought up or removed by choosing **Rule > Gspy**. Forward chaining rules may have Monitor Windows (see **5.8 Debugging with the Environment**) brought up or removed by choosing **Rule > Monitor** (this command is disabled when a backward chaining rule has been selected). These are explained in **5.8 Debugging with the Environment**.

The package used for displaying symbols may be modified by choosing **Works > Tools > Preferences... > Rule Browser > Package** and entering a package name into the Package area. Clicking **OK** will update the tool.
5.8 Debugging with the Environment

5.8.1 Spy Windows

KnowledgeWorks Gspy Window

Spy Windows display graphically the actions or subgoals a rule (either forward or backward chaining) will invoke when it fires. A Spy Window may be obtained by selecting a rule in the Rule Browser and choosing Rule > Gspy or choosing Gspy from the context menu or by choosing Spy > Gspy in the KnowledgeWorks Listener. Spying can be cancelled by closing the Spy Window or by choosing Spy > NoSpy or Spy > NoSpy All from the KnowledgeWorks Listener.

Selecting one of the graph nodes in the top pane of the Spy Window displays the full text of the box in the pane below. Choosing Gspy from the context menu brings up a Spy Window for the goal in the box.

When the rule being displayed fires, execution stops and the buttons at the bottom of the KnowledgeWorks Listener allow the rule to be single-stepped. Clicking on the Creep button steps through the rule, and Leap advances to the end of the rule (unless any of the intervening goals invoke another rule which has been spied). When single-stepping, a highlight marks the action or goal being performed. When execution is suspended in this manner, any of the KnowledgeWorks tools or browsers may be used.

More details on single stepping through rules are in Appendix A: Common Prolog.

5.9 Monitor Windows
Monitor Windows allow the preconditions of forward chaining rules to be monitored. They may be obtained by choosing **Rule > Monitor** or by choosing **Spy > Monitor Rule** from the KnowledgeWorks Listener.

The top part of the window is the **Select instantiations** pane, as described below. The lower part displays a list of either **fired** or **unfired** instantiations. This list is not kept up to date if the rulebase is executing with debugging turned off. To examine a binding in a displayed instantiation, select the corresponding line and choose **Instantiations > Inspect**. This shows the objects themselves in a LispWorks Inspector tool, so double-clicking on one of the entries will cause that entry to be inspected. See the *LispWorks IDE User Guide* for more details.

The **Show Inferencing State** dropdown allows you to choose which named inferencing state is used to find the instantiations.

When the **All Unfired Instantiations** button is selected, the unfired instantiations are displayed.

When the **Matching Selected Conditions** button is selected, the instantiations that match all of the selected preconditions are displayed. The topmost shows the preconditions of the rule. Any conditions that are matched by the object base are highlighted. This highlighting means the condition is matched without reference to any of the other conditions. A message indicates the number of instantiations matching the highlighted preconditions. A group of preconditions matched individually (hence highlighted) may not be matched together if, for instance, variables were bound across them.

If a rule has the conditions, for example:
these would be displayed in the top pane of the Rule Monitor Window. The first two would be highlighted if the object base contained a person object. But instantiations would only be displayed if there was a person object with the same father value as some (other) person object has son.

The selection of conditions may be toggled by left-clicking. So in the above example the last condition could be selected also by clicking on it, and there would be no instantiations displayed if the only consistent value of ?person was nil.

5.9.1 Forward Chaining History

KnowledgeWorks Forward Chaining History

The Forward Chaining History may be viewed by choosing KnowledgeWorks > FC History. This displays the rules which the forward chaining engine has fired. The left pane lists sequentially the contexts which have been executed, with the cycle number in which they were entered. These can be clicked on to show in the right pane, the history for that context. The rules in it are listed down the left, and the cycle numbers along the top, forming a two dimensional grid.

Each position in the grid indicates the status of the rule in that cycle. A colored box indicates that the rule fired. A half-colored box indicates that the rule fired, but that the invocation of the backward chainer on the right-hand side failed at some point. There can only be one colored or half-colored box per cycle. An outlined box indicates that the rule was in the conflict set but was not chosen to fire. Absence of any icon indicates that the rule was not even in the conflict set.

If the forward chaining history is displayed while a rule is executing (for example, while the rule is being single stepped) a half-colored box is displayed as execution is not complete.

The Rule menu can be used in the same way as in the Rule Browser, described in 5.7 The Rule Browser. It applies to the selected rule in the FC Cycles pane.

The Show Inferencing State dropdown allows you to choose which named inferencing state is examined.
This tool is not available when debugging is turned off.
6 Advanced Topics

6.1 Control Flow

6.1.1 Meta Rule Protocol

The meta rule protocol (MRP) reifies the internal actions of the forward chainer in terms of backward chaining goals. This allows the user to debug, modify, or even replace the default behavior of the forward chainer. The basic hooks into the Forward Chaining Cycle provided by the MRP include conflict resolution and rule firing. Each context may have a meta-rule defined for it which behaves as a meta-interpreter for that context. For example, if no meta-rule is defined for a context it behaves as if it were using the following meta-rule:

```lisp
(defrule ordinary-context :backward
  ((ordinary-context)
   <--
   (start-cycle)
   (instantiation ?instantiation)
   (fire-rule ?instantiation)
   (cut)
   (ordinary-context)))
```

This rule describes the actions of the forward chaining cycle for this context. Firstly `start-cycle` performs some internal initializations and updates the conflict set. It is essential that this is called at the start of every cycle. Next the preferred instantiation is selected from the conflict set by the call to `instantiation` and is stored in the variable `?instantiation`. The rule corresponding to this is fired (by `fire-rule`) and the recursive call to `ordinary-context` means that the cycle is repeated. The `cut` is also essential as it prevents back-tracking upon failure. Failure occurs when there are no more instantiations to fire (the `instantiation` predicate fails) and this causes control to be passed on as normal.

A meta-rule may be assigned to a context with the `:meta` keyword of the `defcontext` form. The argument of the `:meta` keyword is the list of actions to be performed by the context. For example, a context using the above ordinary meta-interpreter can be defined by:

```lisp
(defcontext my-context :meta ((ordinary-context)))
```

This implicitly defines the rule:

```lisp
(defrule my-context :backward
  ((my-context)
   <--
   (ordinary-context)))
```

and whenever this context is invoked, the rule of the same name is called. The context could equally well have been defined as:

```lisp
(defcontext my-context :meta
  ((start-cycle)
   (instantiation ?instantiation)
   (fire-rule ?instantiation)
   (cut)
   (my-context)))
```
Sometimes it is useful to manipulate the entire conflict set. For this purpose the action `conflict-set ?conflict-set` will return the entire conflict set in the given variable, in the order specified by the context's conflict resolution strategy. The actions:

```
(conflict-set ?conflict-set)
(member ?instantiation ?conflict-set)
```

are equivalent to:

```
(instantiation ?instantiation)
```

although the latter is more efficient.

Now that the user has access to the instantiations of rules, functions are provided to examine them.

### 6.1.1.1 Functions defined on Instantiations

The following functions may be called on instantiations:

```
(inst-rulename instantiation)
```

which returns the name of the rule of which this is an instantiation.

```
(inst-token instantiation)
```

which returns the list of objects (the `token`) which match the rule. These appear in reverse order to the conditions they match.

```
(inst-bindings instantiation)
```

which returns an a-list of the variables matched in the rule and their values.

### 6.1.1.2 A Simple Example

This meta-rule displays the conflict set in a menu to the user and asks for one to be selected by hand on each cycle. Note that we have to check both that there were some instantiations available, and that the user selected one (rather than clicking on the `Abort` button).

```
(defrule manual-context :backward
  ((manual-context)
   <--
   (start-cycle)
   (conflict-set ?conflict-set)
   (test ?conflict-set)
   ; are there any instantiations?
   ((select-instantiation ?conflict-set)
    ?instantiation)
   (test ?instantiation)
   ; did the user pick one?
   (fire-rule ?instantiation)
   (cut)
   (manual-context)))
```

where the function `select-instantiation` could be defined as:

```
(defun select-instantiation (conflict-set)
  (tk:scrollable-menu conflict-set
```

50
Now a context could be defined by:

(defcontext a-context :strategy ()
  :meta ((manual-context)))

### 6.1.1.3 A Simple Explanation Facility

Meta-rules can also be used to provide an explanation facility. A full implementation of the explanation facility described here is included among the examples distributed with KnowledgeWorks, and is given also in **B.2 Explanation Facility**.

Suppose we have a rule about truck scheduling of the form:

**(defrule allocate-truck-to-load :forward
  (load ?l size ?s truck nil destination ?d location ?loc)
  (test (not (eq ?d ?loc)))
  (truck ?t capacity ?c load nil location ?loc)
  (test (> ?c ?s))
  -->
  (assert (truck ?t load ?l)))
  (assert (load ?l truck ?t)))**

and we wish to add an explanation by entering a form like:

**(defexplain allocate-truck-to-load :why ("~S has not reached its destination
  ~S and ~ does not have a truck
  allocated, ~ ~S does not have a load
  allocated, and ~ with capacity ~S is
  able to carry the load, ~ and both
  are at the same place ~S"
  :what ("~S is scheduled to carry ~S to ~S"
  ?t ?l ?d)
  :because ("A customer requires ~S to be
  moved to ~S" ?l ?d))**

where the **:why** form explains why the rule is allowed to fire, the **:what** form explains what the rule does and the **:because** gives the ultimate reason for firing the rule.

The stages in the implementation are as follows:

- Define a macro called **defexplain** to store the explanation information in, say, a hash-table keyed against the rule name.
- Define a function **add-explanation** takes an instantiation, fetches the explanation information from the hash-table and the variable bindings in the instantiation, and adds the generated explanations to another global data structure, something like:

**:(defun add-instantiation (inst)
  (let ((explain-info
        (gethash (inst-rulename inst)
                 *explain-table*)))
    (when explain-info
      (add-explanation inst))**
• Implement graphical tools to browse the resulting explanations.

• Define a meta-interpreter for which will produce explanations, for example:

  (defrule explain-context :backward
    ((explain-context)
     <--
     (start-cycle)
     (instantiation ?inst)
     ((add-explanation ?inst))
     (fire-rule ?inst)
     (cut)
     (explain-context)))

6.1.1.4 Reasoning with Certainty Factors

Another application of meta-rules is in the manipulation of uncertainty. A full implementation of the uncertain reasoning facility described below is included among the examples distributed with KnowledgeWorks, and also in **B.3 Uncertain Reasoning Facility**.

In this example, we wish to associate a certainty factor with objects in a manner similar to the MYCIN system (see Rule-Based Expert Systems, B. G. Buchanan and E. H. Shortliffe, Addison-Wesley 1984). When we assert an "uncertain" object we wish it to acquire the certainty factor of the instantiation which is firing. We define the certainty factor of an instantiation to be the certainty factor of all the objects making up the instantiation multiplied together. Additionally, we wish rules to have an implication strength associated with them which is a multiplicative modifier to the certainty factor obtained by newly asserted uncertain objects. The general approach is as follows:

• Define global variables *c-factor* to hold the certainty factor of the current instantiation and *implic-strength* to hold the implication strength of the rule, and a class of "uncertain" KnowledgeWorks objects:

  (def-kb-class uncertain-kb-object ()
    (c-factor :initform (* ?c-factor* implic-strength*)
      :accessor object-c-factor))

The uncertain objects should contain this class as a mixin.

• Define a function to obtain the certainty factor of instantiations:

  (defun inst-c-factor (inst)
    (reduce '* (inst-token inst) :key 'object-c-factor))

• Define a conflict resolution tactic to prefer either more or less certain instantiations (See **6.1.2 User-definable Conflict Resolution** for details).

• Define a meta-rule to set the global certainty factor to the certainty factor of the instantiation about to fire:

  (defrule uncertain-context :backward
    ((uncertain-context)
     <--
     (start-cycle)
     (instantiation ?inst)
     ((setq ?c-factor* (inst-c-factor ?inst)))
     (fire-rule ?inst)
     (cut)
     (uncertain-context)))

52
6 Advanced Topics

- Define a function implication-strength which sets the variable *implic-strength* so that rules may set their implication strength by calling the action:

```lisp
((implication-strength <number>))
```

A rule could be defined similarly to:

```lisp
(defrule my-rule :forward
  (my-class ?obj1)
  (my-class ?obj2)
  -->
  ((implication-strength 0.6))
  (assert (my-class ?obj3)))
```

where the certainty factor of the new object ?obj3 will automatically become:

```lisp
(* (object-c-factor ?obj1) (object-c-factor ?obj2) 0.6)
```

While this is an extremely simplistic version of uncertain reasoning, it suggests how a more elaborate treatment might be approached.

### 6.1.2 User-definable Conflict Resolution

A conflict resolution strategy is a list of conflict resolution tactics. A conflict resolution tactic is a function which takes as arguments two rule instantiations, and returns t if and only if the first is preferred to the second, otherwise nil. A conflict resolution tactic may be defined by:

```lisp
(deftactic <tactic-name> {<type>} <lambda-list> [<doc-string> <body>])
```

where `<tactic-name>` is the name of the tactic and of the function being defined which implements it, and `<lambda-list>` is a two argument lambda-list. `<type>` may be either :static or :dynamic, defaulting to :dynamic. A dynamic tactic is one which looks into the objects which match the rule to make up the instantiation; a static one does not. For example, a tactic which prefers instantiations which match, say, truck objects to instantiations which do not could be defined as static. However, if it looks into the slot values of the truck object it should be defined as dynamic. Static tactics are treated more efficiently but wrongly declaring a tactic as static will lead to incorrect conflict resolution. If `doc-string` is given, then it should be a string. The value can be retrieved by calling the function `documentation` with doc-type `function`.

It is an absolute requirement that there exist no instantiations for which:

```lisp
(<tactic-name> <instantiation1> <instantiation2>)
```

and:

```lisp
(<tactic-name> <instantiation2> <instantiation1>)
```

both return t. Consequently, for any single given instantiation:

```lisp
(<tactic-name> <instantiation> <instantiation>)
```

must return nil.

The function which defines a conflict resolution tactic should be computationally cheap as it is used repeatedly and frequently to compare many different pairs of instantiations.
6.1.2.1 Examples

The following tactic prefers instantiations with truck objects to ones without:

(deftactic prefer-trucks :static (inst1 inst2)
  (flet ((truck-p (obj) (typep obj 'truck))
          (and (some #'truck-p (inst-token inst1))
               (notany #'truck-p (inst-token inst2))))))

Note that this tactic would be incorrect if we did not check that the second instantiation does not refer to any trucks (otherwise it would always return t if both instantiations contain trucks). It can safely be declared as static as it does not look into the slots of the objects which make up the instantiation.

This tactic implements alphabetical ordering on rule names:

(deftactic alphabetical-rulename :static (inst1 inst2)
  (string< (symbol-name (inst-rulename inst1))
           (symbol-name (inst-rulename inst2))))

This tactic prefers instantiations which bind the variable ?x to zero:

(deftactic prefer-?x=0 :dynamic (inst1 inst2)
  (flet ((fetch-?x (inst)
          (cdr (assoc '?x (inst-bindings inst))))
          (and (eql 0 (fetch-?x inst1))
               (not (eql 0 (fetch-?x inst2))))))

Note that again we must not forget to check that ?x is not zero in the second instantiation. This tactic must be declared dynamic as ?x must have been instantiated from the slots of one of the matched objects.

The final tactic is for the example of uncertain reasoning and implements a method of preferring "more certain" instantiations:

(deftactic certainty :dynamic (inst1 inst2)
  (> (inst-c-factor inst1) (inst-c-factor inst2)))

This tactic must be dynamic if the certainty factors of objects can be modified after creation. If this is forbidden the tactic could be defined as static. Then the context defined by:

(defcontext my-context :strategy (priority certainty))

will prefer instantiations of rules with higher priority or, if this does not discriminate sufficiently, instantiations which are "more certain".

6.2 Optimization

6.2.1 Forward Chaining

6.2.1.1 KnowledgeWorks Structures

A CLOS class may be replaced by a structure for increased speed when all the power of CLOS is not needed. Within the rule interpreter the structure behaves like a CLOS class which:

- Has an initform of nil for each slot.
6 Advanced Topics

- Has the keyword version of the slot name as initarg for each slot.
- Has only single inheritance.
- Has no methods defined on it.
- Should not be modified from Lisp after its creation.

A KnowledgeWorks structure is defined by the macro:

```
(def-kb-struct <class-spec> <slot-spec>*)
```

where the arguments are the same as for `defstruct` except that in `<class-spec>` only the options :include and :print-function are allowed. A structure may only be included in a KnowledgeWorks structure if it too is a KnowledgeWorks structure defined by `def-kb-struct`. All the functions normally provided by `defstruct` (accessors, a predicate etc.) are generated. An instance of the structure class may be created by the generic function:

```
(make-instance <class-name> {<slot-specifier> <value>)*}
```

where `<slot-specifier>` is the keyword version of the slot name, as with any structures, and `<value>` is the value the slot is to take, otherwise defaulting to the value specified in the `def-kb-struct` form. If created from Lisp by any means other than `make-instance` (for example, by the automatically defined `make-<structure-name>` constructor), the inference engine will not know about the structure.

Once created, structures must not be modified directly from Lisp as this will corrupt the state of the forward chaining inference engine. For example:

```
(def-kb-struct train position speed)
(def-kb-struct signal position color)
(make-instance 'train :position 0 :speed 80)
(make-instance 'signal :position 10 :color 'red)
```

defines KnowledgeWorks structures for trains and signals and makes an instance of each. Note that they are not fully-fledged CLOS objects but are analogous to working memory elements in OPS5.

### 6.2.1.2 Efficient Forward Chaining Rule Preconditions

Forward chaining rules are more efficient if the more restrictive preconditions (that is, the ones which will have fewer matches) are written first. Computationally cheap Lisp tests should be used wherever possible as they reduce the search space of the rule interpreter. The Lisp tests should where possible be broken into sufficiently small pieces that they can be applied as early on as possible.

For example, the precondition fragment:

```
(train ?t position ?p1)
(test (> ?p1 5))
(signal ?s position ?p2)
(test (> ?p2 6))
```

is better than:

```
(train ?t position ?p1)
(signal ?s position ?p2)
(test (and (> ?p1 5) (> ?p2 6)))
```

because in the first example the Lisp tests can be applied directly to the trains and signals respectively before looking at
combinations of trains and signals, whereas in the second case all the combinations must be produced before the Lisp test can be applied. Simply separating the tests is enough for the rule compiler to apply them to the right object base matches — the precise order of the tests is unimportant.

6.2.1.3 Profiling


6.2.2 Conflict Resolution

6.2.2.1 Use of Contexts

The single most significant way to improve conflict resolution time is to divide the rulebase up into contexts. The time taken by conflict resolution is dependent on the total number of instantiations of all the rules in the context so the fewer rules in each context, the more efficient conflict resolution will be.

6.2.2.2 Optimization of the Strategy

A conflict resolution strategy may be optimized by combining the constituent tactics in a more effective manner. There are three different types of conflict resolution tactic:

- **Rule-defined** (meaning the tactic relies only on the rule of the instantiation and on nothing else), including priority, -priority, order, -order, specificity and -specificity.
- **Static** (meaning the tactic does not look into the slots of the matched objects which make up the instantiation), including recency and -recency.
- **Dynamic** (meaning the tactic may look into the objects making up the instantiation), including mea, -mea, lex and -lex.

KnowledgeWorks is best able to optimize rule-defined tactics and least able to optimize dynamic tactics. The optimizations for a particular type of tactic can only be applied if it is preceded only by tactics which can be optimized to the same degree (or better). For example, in the strategy (recency priority), the tactic priority would only be optimized as a static tactic. In the strategy (priority mea recency), priority can be optimized as a rule-defined tactic but recency will be treated as a dynamic tactic.

Some final points to bear in mind:

- Tactics which tend to prefer existing instantiations over newer ones (for example -mea, -lex and -recency) will degrade performance.
- recency and lex have similar functionality but recency is more efficient.

6.2.3 Backward Chaining

6.2.3.1 Pattern Matching

The KnowledgeWorks Backward Chainer indexes clauses for a backward rule based on the first argument. If the first arguments to backward rule clauses are distinct non-variables, the backward chainer can pre-select possible matching clauses for a call.

For example, in the following rule:
6.2.3.2 Tail Recursion

The KnowledgeWorks Backward Chainer supports the transformation of "tail-recursive" calls into jumps. Thus, stack overflow can be avoided without resorting to "repeat, fail" loops in most cases. For example, given the definition:

```lisp
(defrule run-forever :backward
  ((run-forever)
   (run-forever)))
```

the call: `(run-forever)` will run forever without generating a stack overflow. Note that this optimization is not limited to recursive calls to the same rule. The last call of any rule will be compiled as a jump, drastically reducing stack usage.

6.2.3.3 Cut

The use of "cut" is a well known performance enhancement for Prolog-style rules. In KnowledgeWorks it does more than reduce the time spent in search. When a "cut" is invoked, all the stack space between the initial call to the containing rule and the current stack location is reclaimed immediately, and can have a significant impact on the total space requirements of a program.

6.3 Use of Meta-Classes

Objects of meta-classes other than `standard-class` may be made available to KnowledgeWorks by including the KnowledgeWorks mixin `standard-kb-object`. This requires:

- The existence of a `validate-superclass` method allowing `standard-kb-object` (meta-class `standard-class`) to be a superclass of the class being defined with a different meta-class.
- That the meta-class in question does not implement any particularly strange behavior on slot access, for example, if querying a slot value results in setting it.

6.3.1 Example

A meta-class `standard-kb-class` could be defined as a KnowledgeWorks class. New KnowledgeWorks classes (or even ordinary non-KnowledgeWorks classes) could be defined with this meta-class. KnowledgeWorks could then reason about the instances of the classes and about the class objects themselves. The code below implements this:

```lisp
(def-kb-class standard-kb-class (standard-class) ()
  (defmethod validate-superclass
   ((class standard-kb-class)
    (superclass standard-class))
   t)
  (def-kb-class foo () ((slot))
   (:metaclass standard-kb-class))
```
Then when the following rule fires:

```lisp
(defrule find-kb-class :forward
  (standard-kb-class ? clos::name ?n)
  -->
  ((format t "I can reason about class ~s" ?n)))
```

it will output:

```
I can reason about class FOO
```

### 6.4 Logical Dependencies and Truth Maintenance

When a rule creates an object that depends on a specific set of preconditions, it is sometimes necessary to erase that object when those preconditions no longer hold. This is an example of truth maintenance.

KnowledgeWorks provides a mechanism to track logical dependencies between objects and preconditions which cause any dependent objects to be erased automatically. This is achieved using a **logical** clause in a forward chaining rule, with a precondition of the form:

```
(logical <forward-condition>+)
```

The enclosed forward conditions in this clause are matched as normal, but if the rule fires and creates new objects (by `assert` or `make-instance`) then these objects are associated with the enclosed conditions. If the conditions are found to be false in the future, then the created objects are erased automatically (see `erase`).

**NB:** There can be at most one logical clause in a rule (though it can contain multiple subclauses) and it must be the first clause in the rule. Other clauses can follow the logical clause, but they are not part of the logical dependency.

### 6.4.1 Example

Given the following classes and rules:

```lisp
(def-kb-class number-object ()
  ((value :initarg :value)))
(def-kb-class have-some-large-numbers ()
  ()
)
(defrule notice-a-large-number :forward
  (logical (number-object ? value ?value)
    (test (> ?value 100)))
  -->
  (assert (have-some-large-numbers ?)))
```

then a `have-some-large-numbers` object will be created when a number larger than 100 exists:

```lisp
(setq n1 (make-instance 'number-object :value 10))
(infer)
(any '?x '(have-some-large-numbers ?x)) ==> false
(setf (slot-value n1 'value) 200) ; this is large
(infer)
(any '?x '(have-some-large-numbers ?x)) ==> true
```

In addition, when the large number becomes smaller, the `have-some-large-numbers` object will be erased again:
(setf (slot-value n1 'value) 55)
(infer)
(any '?x '(have-some-large-numbers ?x)) ==> false

because a logical dependency was tracked between the preconditions:

(number-object ? value ?value)
(test (> ?value 100)

and the have-some-large-numbers object.

6.5 Inferencing States

An inferencing state represents all the state needed to run the forward chaining interpreter, including the object base, the current cycle number and the set of unfired instantiations. It does not include rule or context definitions or any backward chaining state information.

6.5.1 Creating and Maintaining Inferencing States

Inferencing states are first-class objects that can be created and destroyed as required. Each inferencing state must have a unique name (as compared with eql) and initially there is a single inferencing state named :default.

The function make-inferencing-state makes a new empty inferencing state. Inferencing states must be destroyed with destroy-inferencing-state when no longer needed, to release the memory that they use.

Inferencing states can be found using the function find-inferencing-state and the function list-all-inferencing-states can be used to make a list of all known inferencing states.

6.5.2 The Current Inferencing State

The value of the variable *inferencing-state* is known as the current inferencing state. Its value can be changed before calling KnowledgeWorks functions, but should not be changed within the body of a rule.

Some operations, such as object creation, slot modification, reset and infer only affect the current inferencing state. Backward chaining operations that match the object base only find objects from the current inferencing state.

Operations that change rules or contexts, such as defrule and clear-all, affect all inferencing states.

6.5.3 Uses of Inferencing States

In many cases, a single inferencing state is sufficient and the initial inferencing state named :default can be used without any special effort.

To allow several independent inferencing operations to be performed simultaneously, multiple inferencing states must be managed explicitly. Some typical situations are described below.

6.5.3.1 Multiple threads

By binding *inferencing-state* around all KnowledgeWorks operations in a thread's main function as in the example below, its value can be unique to each thread.

(defun test-1-counter (name)
  (let* ((*inferencing-state* nil)
         (step (1+ (random 10))))
    ...)

6 Advanced Topics
(limit (* step (+ 2000 (random 100))))) (unwind-protect (progn (setq *inferencing-state* (make-inferencing-state name)) (make-instance 'counter :value limit :step step) (infer)) (destroy-inferencing-state *inferencing-state*)))

(mp:process-run-function (format nil "Test ~D" index) '() 'test-1-counter (gensym)

6.5.3.2 Interleaved in a Single Thread

By binding *inferencing-state* around specific KnowledgeWorks operations in a function as in the example below, multiple inferencing states can be maintained within a single thread.

(defun test-stepping-single-context () (let ((state1 (make-inferencing-state 'state1)) (state2 (make-inferencing-state 'state2))) (unwind-protect (progn (let (((*inferencing-state* state1)) (make-instance 'step-controller :kb-name 'stepper-one-a)) (let (((*inferencing-state* state2)) (make-instance 'step-controller :kb-name 'stepper-one-b)) (loop repeat 10 do (let (((*inferencing-state* state1)) (infer))) (let (((*inferencing-state* state2)) (infer)))) (destroy-inferencing-state state1) (destroy-inferencing-state state2))))
7 Reference Guide

The symbols documented in the following pages are all external in the KW package unless stated otherwise. They are listed in alphabetical order.

---

**all-debug**

*Function*

**Summary**

Turns debugging facilities on.

**Package**

`kw`

**Signature**

`all-debug`

**Description**

The function `all-debug` turns on all KnowledgeWorks debugging facilities. This means that rules and contexts can be single stepped and monitored, and a record is kept of whenever objects are created or modified.

This should be called before compiling any rules or contexts that are to be debugged.

**Examples**

```
(all-debug)
```

**See also**

`no-debug`

---

**any**

*Function*

**Summary**

Return the first match of a backward chaining goal.

**Package**

`kw`
Signature

\textbf{any} \ pattern-to-instantiate \ goal-to-prove => result, successp

Arguments

\textit{pattern-to-instantiate}↓

A list or symbol.

\textit{goal-to-prove}↓

Any backward chaining goal.

Values

\textit{result} \; nil or a value matching \textit{pattern-to-instantiate}.

\textit{successp} \; A boolean.

Description

The function \textit{any} starts the backward chaining inference engine to look for any set of bindings which satisfy \textit{goal-to-prove}. Using those bindings, \textit{pattern-to-instantiate} is instantiated and returned.

Two values are returned. The second value indicates with \texttt{t} that a proof was found, or with \texttt{nil} that no proof exists. In the former case, the first value is the instantiated version of \textit{pattern-to-instantiate}, in the latter case, the first value is \texttt{nil}.

Any subgoals that match the object base will only find objects from the current inferencing state.

Examples

\begin{verbatim}
(any '(?x is in (1 2 3)) '(member ?x (1 2 3)))
\end{verbatim}

returns \texttt{(1 IS IN (1 2 3)), T}.

\begin{verbatim}
(any '(?truck is a truck) '(truck ?truck))
\end{verbatim}

returns \texttt{(#<TRUCK TRUCK5> IS A TRUCK), T}.

See also

\textit{findall}

\textbf{assert} \\
\textit{Backward Chaining Goal}

Summary

Creates or modifies objects in the object base.

Package

\textit{kw}

Signature

\textbf{assert} \; (class-name \; variable \; \{slot-and-term\}*)
7 Reference Guide

slot-and-term ::= (slot-name term)

Arguments

class-name⇩

A variable beginning with ?.

variable⇩

The name of a class.

class-name

The name of a slot in class-name.

slot-name⇩

An expression.

term⇩

Description

The backward chaining goal assert creates or modifies objects in the object base.

class-name must be the name of a class of objects known to KnowledgeWorks. Each term is an expression composed of Lisp data structures and KnowledgeWorks variables.

If variable is unbound a new instance of class-name is created with each named slot-name initialized to the value of the corresponding term.

If variable is bound, that bound instance has its named slots modified to contain the values of term corresponding to each slot-name. It is an error if the bound object is not of the named class.

It is an error to put an unbound variable into a slot of an object in the object base.

Only objects in the current inferencing state will be affected.

Examples

(assert (truck ?truck driver ?driver))
(assert (possible-trucks ? trucks (?truck . ?trucks))

See also

erase

clear-all

Function

Summary

Clears all contexts, rules and objects.

Package

kw

Signature
clear-all

description

The function clear-all clears all contexts, rules and objects. The list of KnowledgeWorks classes remains unaffected. The
default context `default-context` is not removed, but all rules in it are.

The function affects all inferencing states.

Examples

```(clear-all)```

See also

- `clear-rules`
- `reset`

### clear-rules

**Function**

**Summary**

Clears all contexts and rules.

**Package**

`kw`

**Signature**

`clear-rules`

**Description**

The function `clear-rules` clears contexts and rules. The list of KnowledgeWorks classes and the object base remains unaffected. The default context `default-context` is not removed, but all rules in it are.

This function affects all inferencing states.

Examples

```(clear-rules)```

See also

- `clear-all`
- `reset`

### conflict-set

**Backward Chaining Goal**

**Summary**

Finds the current meta-interpreter rule instantiations.
Package

\texttt{kw}

Signature

\texttt{conflict-set variable}

Arguments

\texttt{variable} \downarrow \quad \text{An unbound KnowledgeWorks variable introduced by \texttt{?}.}

Description

The backward chaining goal \texttt{conflict-set} is only relevant when writing a meta-interpreter for a context. \texttt{conflict-set} binds \texttt{variable} to the list of all existing rule instantiations in the currently executing context. This list is in the order preferred by the conflict resolution strategy for the context.

Examples

\texttt{(conflict-set ?conflict-set)}

See also

\texttt{instantiation}

\texttt{fire-rule}

---

\textbf{context}

\textit{Backward Chaining Goal}

Summary

Adds new contexts to the agenda.

Package

\texttt{kw}

Signature

\texttt{context context-list}

Arguments

\texttt{context-list} \downarrow \quad \text{A list of context names.}

Description

The backward chaining goal \texttt{context} adds new contexts \texttt{context-list} on top of the agenda (the context stack). The current context is not changed. It is an error if the named contexts do not exist.

If \texttt{context-list} contains variables, then they must be already bound.
**current-cycle**

Function

Summary

Returns the current forward chaining cycle number.

Package

kw

Signature

\[
\text{current-cycle} \Rightarrow \text{cycle-number}
\]

Values

\[
\text{cycle-number} \quad \text{An integer.}
\]

Description

The function \text{current-cycle} returns the current cycle number of the forward chaining rule interpreter in the current inferencing state. If the forward chaining rule interpreter is not running, then it returns the total number of cycles executed by the forward chaining rule interpreter the last time it ran. If the forward chaining rule interpreter has not run at all, then it return zero.

See also

*inferencing-state*

---

**cut**

Backward Chaining Goal

Summary

The standard prolog predicate that stops backtracking.

Package

kw

Signature

cut
Description

The backward chaining goal `cut` is a standard prolog predicate. When first called it succeeds and freezes certain choices made by the backward chainer up to this point. It may no longer attempt to resatisfy any of the goals between the start of clause and the `cut`, and it may not attempt to use any other clauses to satisfy the same goal.

Examples

```prolog
(defrule nice :backward
  ((nice ?x)
   <---
   (rottweiler ?x)
   (cut)
   (fail))
  ((nice ?x) <-->))
```

implements "everything is nice unless it is a rottweiler". First the backward chainer will attempt to prove `(nice fido)` with the first clause. If `fido` is a rottweiler the `cut` then prevents the backward chainer from using the second clause which says "everything is nice". The fail ensures that `(nice fido)` fails.

See also

`fail`

`*cycle*`  

* Symbol Macro

Summary

Deprecated.

Package

`kw`

Description

The symbol macro `*cycle*` is deprecated. New code should use `current-cycle`.

Prior to LispWorks 5.0, `*cycle*` was a variable.

See also

`current-cycle`

`defcontext`  

* Macro

Summary

Defines a context.
7 Reference Guide

Package
kw

Signature
defcontext context-name &key refractoriness auto-return strategy meta documentation

Arguments
class-name

The name of the context being defined.
refractoriness

A boolean.
auto-return

A boolean.
strategy

A list of symbols.
meta

A list of actions.
documentation

A string.

Description

The macro `defcontext` defines a context named `context-name`. If a context of that name already exists then it, and all the rules in it, are first removed.

If `refractoriness` is `nil` then a rule instantiation remains eligible to fire again after firing once. If `refractoriness` is `t` (the default) then each rule instantiation will only fire once.

`auto-return` indicates, when there are no more rules to be fired in the context, whether to signal an error or simply to pass control to the next context on the agenda. The default value `t` passes control on without an error.

`strategy` is the conflict resolution strategy for the context, consisting of a list of tactic names.

`meta` is a list of actions which make up the optional meta-interpreter for the context.

If `documentation` is supplied, then it should be a string. The value can be retrieved by calling the function `documentation` with doc-type `context`.

Examples

```clojure
(defcontext my-context :strategy (priority recency))
(defcontext another-context :strategy (order)
  :meta ((start-cycle)
    (instantiation ?inst)
    (fire-rule)
    (cut)
    (another-context)))
```

See also

`standard-context`
- `lex`
- `lex`
- `mea`
- `mea`
- `order`
- `order`
def-kb-class	Macro

Summary
Defines a class for use in the object base.

Package
kw

Signature
def-kb-class class-name superclass-list slot-descriptions &rest options => class

Arguments
class-name⇑ A symbol.
superclass-list⇑ A list of symbols.
slot-descriptions⇑ A list of defclass slot descriptions.
options⇑ defclass options.

Values
class The named class object.

Description
The macro def-kb-class defines a new CLOS class name class-name, as defclass does. However, if none of the superclasses in superclass-list is a subclass of standard-kb-object, then standard-kb-object is added to the list of superclasses.

slot-descriptions and options are used as in the standard defclass macro.

Examples

(def-kb-class vehicle () ((driver :initarg :driver)))
(def-kb-class truck (vehicle)
  ((load :accessor truck-load)))

See also

def-named-kb-class
def-kb-struct
def-kb-struct

Macro

Summary
Defines a structure class for use in the object base.

Package
kw

Signature

\[
\text{def-kb-struct } \text{name-and-options} \ (\text{slot-description})* \Rightarrow \text{name}
\]

\[
\text{name-and-options} ::= \text{name} \mid (\text{name} \ \text{option})*
\]

\[
\text{option} ::= (:\text{include} \ \text{superclass}) \mid (:\text{print-function} \ \text{print-function})
\]

Arguments

\[
\text{slot-description} \-downarrow \quad \text{A } \text{defstruct} \text{ slot description.}
\]

\[
\text{name} \downarrow \quad \text{A symbol.}
\]

\[
\text{superclass} \downarrow \quad \text{A symbol.}
\]

\[
\text{print-function} \downarrow \quad \text{A symbol or a lambda expression.}
\]

Values

\[
\text{name} \quad \text{The name of the structure class.}
\]

Description

The macro def-kb-struct defines a KnowledgeWorks structure class name name. Objects of these classes are analogous to Lisp structures except that they may be used in rules similarly to CLOS objects.

If superclass is supplied then name will inherit from superclass, which must be KnowledgeWorks structure class.

print-function and slot-description are used as in defstruct.

Examples

\[
(\text{def-kb-struct } \text{start})
\]

\[
(\text{def-kb-struct } \text{named-kb-struct}
\quad (:\text{print-function} \ \text{print-named-kb-struct}))
\quad (\text{name} \ (\text{gensym} \ \text{'named-kb-struct}))
\]

\[
(\text{def-kb-struct } \text{possible-trucks-for-load}
\quad (:\text{include} \ \text{named-kb-struct})
\quad \text{load trucks})
\]
See also
def-kb-class

def-named-kb-class

Summary
Defines a class of named objects for use in the object base.

Package
kw

Signature
def-named-kb-class  class-name  superclass-list  slot-descriptions  &rest  options  =>  class

Arguments
class-name\  A symbol.
superclass-list\  A list of symbols.
slot-descriptions\  A list of defclass slot descriptions.
options\  defclass options.

Values
class\  The named class object.

Description
The macro def-named-kb-class defines a new CLOS class name class-name, as defclass does. However, if none of the superclasses in superclass-list is a subclass of named-kb-object, then named-kb-object is added to the list of superclasses. The class inherits a name slot called kb-name, with accessor kb-name and default initialization form (:initform) that generates a symbol from the class name using (gentemp class-name).

slot-descriptions and options are used as in the standard defclass macro.

Examples

(def-named-kb-class vehicle ()
  ((driver :initarg :driver)))
(def-named-kb-class truck (vehicle)
  ((load :accessor truck-load)))

See also
def-kb-class
def-kb-struct
get-kb-object
kb-name
named-kb-object
Summary
Defines a rule.

Package
kw

Signature
\texttt{defrule \texttt{rule-name} \texttt{direction} \&optional \texttt{doc-string} \&body \texttt{body} \Rightarrow \texttt{rule-name}}

Arguments
\begin{itemize}
  \item \texttt{rule-name} \(\Rightarrow\) A symbol.
  \item \texttt{direction} \(\Rightarrow\) Either \texttt{:forward} or \texttt{:backward}.
  \item \texttt{doc-string} \(\Rightarrow\) An optional string.
  \item \texttt{body} \(\Rightarrow\) Forms as described in \texttt{3 Rules}.
\end{itemize}

Values
\begin{itemize}
  \item \texttt{rule-name} A symbol.
\end{itemize}

Description
The macro \texttt{defrule} defines a rule named \texttt{rule-name} (which must be distinct from any other rule name, context name or KnowledgeWorks class name). If \texttt{direction} is \texttt{:forward} a forward chaining rule is defined, if \texttt{:backward} a backward chaining rule is defined. If \texttt{doc-string} is given, then it should be a string. The value can be retrieved by calling the function \texttt{documentation} with doc-type \texttt{rule}.

A full description of \texttt{body} is given in \texttt{3 Rules}.

Examples
\begin{verbatim}
(defrule move-train :forward :context trains
  (train ?train position ?train-pos)
  (signal ?signal position ?signal-pos color green)
  (test (= ?signal-pos (1+ ?train-pos)))
  --->
  (format t "Train moving to ~S" ?signal-pos)
  (assert (signal ?signal color red))
  (assert (train ?train position ?signal-pos))
(defrule link-exists :backward
  ((link-exists ?town1 ?town2)
   <--
   (or (link ?link town1 ?town1 town2 ?town2)
    (link ?link town2 ?town1 town1 ?town2))
    (cut))
  ((link-exists ?town1 ?town2)
   <--
    (route-exists ?town1 ?town2)))
\end{verbatim}
deftactic

Summary
Defines a tactic function for use in context strategies.

Package
kw

Signature
deftactic tactic-name type lambda-list &body body => tactic-name

Arguments
tactic-name⇓ A symbol.
type⇓ Either :static or :dynamic.
lambda-list⇓ A two argument lambda list.
body⇓ A function body.

Values
tactic-name A symbol.

Description
The macro deftactic defines a new conflict resolution tactic named tactic-name.

type is the type of the tactic, which may be :static if body does not look into the slots of the objects making up the instantiation, otherwise :dynamic.

lambda-list specifies two variable, which will be bound to two instantiation objects and when the forms of body are evaluated. body should return non-nil if and only if the first instantiation object is preferred to the second.

deftactic also defines a function named tactic-name and body can be preceded by a documentation string.

The newly defined tactic may be used as any in-built tactic.

Examples

(defun prefer-trucks (inst1 inst2)
  (let ((truck-p (obj) (typep obj 'truck)))
    (and (some #'truck-p (inst-token inst1))
         (notany #'truck-p (inst-token inst2))))

The new tactic may be used in a defcontext form:

(defcontext my-context :strategy (prefer-trucks))
See also

- `inst-bindings`
- `inst-token`
- `inst-rulename`
- `defcontext`

**destroy-inferencing-state**

**Function**

**Summary**

Destroys an inferencing state.

**Package**

`kw`

**Signature**

`destroy-inferencing-state name-or-state`

**Arguments**

`name-or-state`\da Any object.

**Description**

The function `destroy-inferencing-state` destroys an inferencing state named by `name-or-state`. If `name-or-state` is an inferencing state, then it is destroyed. Otherwise, any inferencing state with that name (as compared using `eql`) is destroyed.

It is an error to destroy the current inferencing state.

**Examples**

```
(destroy-inferencing-state 'my-state)
```

See also

- `find-inferencing-state`
- `*inferencing-state*`
- `inferencing-state-name`
- `list-all-inferencing-states`
- `make-inferencing-state`
**erase**

*Summary*

Erases an object from the object base.

*Package*

`kw`

*Signature*

`erase variable`

*Arguments*

`variable⇓ A` a KnowledgeWorks object.

*Description*

The backward chaining goal `erase` erases an object from the object base. `variable` must be bound to a KnowledgeWorks CLOS object or a KnowledgeWorks structure. The given object is removed from the object base of the current inferencing state.

*Examples*

```
(erase ?x) ; ?x bound to an object
```

*See also*

`assert`

**fail**

*Summary*

The standard prolog predicate that always fails.

*Package*

`kw`

*Signature*

`fail`
Description

The backward chaining goal fail always fails. It is sometimes used with cut.

Examples

```
(defrule nice :backward
  ((nice ?x)
    <--
    (rottweiler ?x)
    (cut)
    (fail))
  ((nice ?x) <--))
```

implements "everything is nice unless it is a rottweiler".

See also

cut

```
findall
findallsset
```

Summary

Return all matches of a backward chaining goal.

Package

kw

Signatures

```
findall pattern-to-instantiate goal-to-prove => list
findallsset pattern-to-instantiate goal-to-prove => set
```

Arguments

```
pattern-to-instantiate
  A list or symbol.
goal-to-prove
  Any backward chaining goal.
```

Values

```
list
  A list.
set
  A list.
```

Description

The function findall starts the backward chaining inference engine to look for all sets of bindings which satisfy goal-to-prove. For each of those bindings, pattern-to-instantiate is instantiated and collected to return a list. The value is nil if
nothing goal-to-prove cannot be satisfied.

Any subgoals that match the object base will only find objects from the current inferencing state.

The function `findallset` is like `findall` but `set` will not have any duplicates (as compared by `equal`).

Examples

```lisp
(findall '?x is in (1 2 3) 'member ?x (1 2 3))
```
returns:

```lisp
((1 IS IN (1 2 3))
 (2 IS IN (1 2 3))
 (3 IS IN (1 2 3)))
```

```lisp
(findall '?truck is a truck) 'truck ?truck)
```
returns:

```lisp
((#<TRUCK TRUCK1> IS A TRUCK)
 (#<TRUCK TRUCK2> IS A TRUCK))
```

See also

`any`

---

**find-inferencing-state**

*Function*

**Summary**

Finds a known inferencing state.

**Package**

`kw`

**Signature**

`find-inferencing-state name &key if-does-not-exist => state`

**Arguments**

- `name`:
  - Any object.

- `if-does-not-exist`:
  - Either `:error` or `:create`.

**Values**

- `state`:
  - An inferencing state.
Description

The function \texttt{find-inferencing-state} finds and returns an inferencing state named by \textit{name}.

If an inferencing state with the same name already exists (as compared using \texttt{eql}), it is returned.

Otherwise, the value of \textit{if-does-not-exist} determines what happens:

- \texttt{:error} A continuable error is signaled. Invoking the \texttt{continue} restart creates and returns a new inferencing state.
- \texttt{:create} A new inferencing state is created and returned.

Examples

\begin{verbatim}
(find-inferencing-state 'my-state)
\end{verbatim}

See also

\texttt{destroy-inferencing-state} \\
\texttt{*inferencing-state*} \\
\texttt{inferencing-state-name} \\
\texttt{list-all-inferencing-states} \\
\texttt{make-inferencing-state}

\begin{tabular}{|l|}
\hline
\textbf{fire-rule} \hspace{1cm} \textit{Backward Chaining Goal} \\
\hline
\end{tabular}

Summary

Fires the given meta-interpreter rule instantiation.

Package

\texttt{kw}

Signature

\texttt{fire-rule instantiation}

Arguments

\begin{verbatim}
instantiation \hspace{1cm} An instantiation object.
\end{verbatim}

Description

The backward chaining goal \texttt{fire-rule} is only relevant when writing a meta-interpreter for a context. \texttt{fire-rule} fires the given rule instantiation \textit{instantiation}. It is an error if the passed object is not an instantiation object.

Examples

\begin{verbatim}
(fire-rule ?instantiation)
\end{verbatim}
**get-kb-object**  
*Function*

**Summary**
Finds a named object in the object base.

**Package**
*kw*

**Signature**
get-kb-object  object-name => object

**Arguments**
object-name ⇩ A symbol.

**Values**
object A KnowledgeWorks CLOS object.

**Description**
The function get-kb-object returns the KnowledgeWorks object named object-name in the object base of the current inferencing state. If there is no such object an error results.

Classes of named objects can be defined using the macro def-named-kb-class.

**Examples**

`(get-kb-object 'fred)`

**See also**
def-named-kb-class
kb-name
**infer**

*Function*

**Summary**

Runs the forward chaining inferencing engine.

**Package**

`kw`

**Signature**

`infer &key contexts => cycle-count`

**Arguments**

`contexts` A list of context names, default to `(default-context)`.

**Values**

`cycle-count` An integer.

**Description**

The function `infer` runs the forward chaining inference engine in the current inferencing state, with `contexts` as the initial agenda. The first rules to fire will be from the first context listed in `contexts` until control is passed on. The value returned as `cycle-count` is the total number of cycles executed (given in `current-cycle`).

**Examples**

```
(infer :contexts '(my-context another-context))
```

**See also**

`current-cycle`

---

**:*inferencing-state*:**

*Variable*

**Summary**

The current inferencing state.

**Package**

`kw`

**Initial Value**

An empty inferencing state named `:default`. 
Description

The value of the variable *inferencing-state* is the current inferencing state for many KnowledgeWorks functions. This variable can be bound to a particular inferencing state before calling other KnowledgeWorks functions, but should not be changed within the body of a rule.

See also

current-cycle
destroy-inferencing-state
find-inferencing-state
inferencing-state-name
list-all-inferencing-states
make-inferencing-state

**inferencing-state-name**  
*Function*

Summary

Returns the name of an inferencing state.

Package

kw

Signature

**inferencing-state-name**  
*state* =>  *name*

Arguments

*state*  
An inferencing state.

Values

*name*  
Any object.

Description

The function **inferencing-state-name** returns the name of *state*.

Examples

  (inferencing-state-name *inferencing-state*)

See also

find-inferencing-state
*inferencing-state*
list-all-inferencing-states
make-inferencing-state
*in-interpreter*  

**Variable**

**Summary**  
Allows code to detect when it is running in a rule.

**Package**  
kw

**Initial Value**  
nil

**Description**  
The variable *in-interpreter* is bound to \texttt{t} if the code executing has been called (directly or indirectly) from the forward chaining rule interpreter. Otherwise it bound to \texttt{nil}. The value should not be changed.

---

\textbf{instantiation}  

**Backward Chaining Goal**

**Summary**  
Find the next meta-interpreter rule instantiation that will fire.

**Package**  
kw

**Signature**  
\texttt{instantiation variable}

**Arguments**  
\texttt{variable}\

An unbound variable introduced by \texttt{?}.

**Description**  
The backward chaining goal \texttt{instantiation} is only relevant when writing a meta-interpreter for a context. \texttt{instantiation} binds \texttt{variable} to the next preferred instantiation from the conflict set of the currently executing context.

This goal may be satisfied repeatedly each time returning the next instantiation. When no instantiations are left, it fails.

**Examples**

\begin{verbatim}
(instantiation ?instantiation)
\end{verbatim}
inst-bindings

Function

Summary

Returns the bindings in a rule instantiation.

Package

kw

Signature

inst-bindings instantiation => bindings

Arguments

instantiation An instantiation object.

Values

bindings An association list.

Description

The function inst-bindings returns an association list of the variables and their bindings in instantiation. The variables are those produced by the condition part of the forward chaining rule.

Examples

For an instantiation of a rule with the precondition:

\[(\text{object} \ ? \ \text{color} \ ?\text{color-value} \ \text{size} \ ?\text{size})\]

the value returned by:

\[(\text{inst-bindings} \ \text{inst})\]

might be:

\[\{(?\text{color-value} . :\text{red}) \ (?\text{size} . 20)\}\]
See also

conflict-set
deftactic
inst-rulename
inst-token
instantiation

**inst-rulename Function**

Summary

Returns the rule name of a rule instantiation.

Package

kw

Signature

\[ \text{inst-rulename } \text{instantiation} \Rightarrow \text{rulename} \]

Arguments

\[ \text{instantiation} \]

An instantiation object.

Values

\[ \text{rulename} \]

A symbol which is the name of a rule.

Description

The function \text{inst-rulename} returns the rule name of \text{instantiation} (the name of the rule of which this is an instantiation).

See also

conflict-set
inst-bindings
deftactic
inst-token
instantiation

**inst-token Function**

Summary

Returns the token of a rule instantiation.

Package

kw
Signature

\texttt{inst-token \ instantiation} => \texttt{token}

Arguments

\texttt{instantiation} \quad \text{An instantiation object.}

Values

\texttt{token} \quad \text{A list of objects.}

Description

The function \texttt{inst-token} returns the token of \texttt{instantiation}. \texttt{token} is the list of objects that match the condition part of the forward chaining rule. This list of objects is in reverse order to the order in which the conditions appear in the rule.

Examples

If the forward chaining conditions are:

\[
\begin{align*}
\text{(train ?train)} \\
\text{(signal ?signal)}
\end{align*}
\]

then the token will have the form \texttt{(signal-object train-object)}.

See also

\texttt{conflict-set}  \\
\texttt{deftactic}  \\
\texttt{inst-rulename}  \\
\texttt{inst-bindings}  \\
\texttt{instantiation}

\textbf{kb-name} \quad \textit{Generic Function}

Summary

Returns the name of an object.

Package

\texttt{kw}

Signature

\texttt{kb-name object} => \texttt{name}

Arguments

\texttt{object} \quad \text{A KnowledgeWorks named CLOS object.}
Values

\textit{name} \hspace{1cm} A symbol.

Description

The generic function \texttt{kb-name} returns the name of \textit{object}. It is an error if \textit{object} is not a named object. Classes of named objects can be defined using the macro \texttt{def-named-kb-class}.

Examples

\begin{verbatim}
(kb-name (get-kb-object 'fred)) ; returns FRED
\end{verbatim}

See also

\texttt{def-named-kb-class}
\texttt{get-kb-object}
\texttt{named-kb-object}

\section*{kw-class}

\textit{Backward Chaining Goal}

Summary

Matches all KnowledgeWorks class names.

Package

\texttt{kw}

Signature

\texttt{kw-class term}

Arguments

\begin{verbatim}
term
\end{verbatim}

Any backward chaining term.

Description

The backward chaining goal \texttt{kw-class} matches all KnowledgeWorks class names. It can act as a generator and can be resatisfied. It succeeds when \textit{term} is a symbol which is the name of a KnowledgeWorks class. If \textit{term} is an unbound variable it generates the names of the KnowledgeWorks classes.

Examples

\begin{verbatim}
(kw-class truck) ; succeeds if truck is a KW class

(kw-class ?class)
 ; ?class is bound to the name of a KW class
\end{verbatim}
See also

def-kb-class
def-kb-struct
def-named-kb-class

lex

Conflict Resolution Tactic / Function

Summary

Implements the lex tactic.

Package

kw

Signature

lex instantiation1 instantiation2 => result

Arguments

instantiation1↓ An instantiation object.
instantiation2↓ An instantiation object.

Values

result A boolean.

Description

The conflict resolution tactic / function lex implements the LEX tactic. It returns true if and only if instantiation1 is preferred to instantiation2 by the conflict resolution tactic lex, otherwise false. The function is intended to be used primarily by including it in the conflict resolution strategy for a context.

Examples

(defcontext my-context1 :strategy (lex))
(defcontext my-context2 :strategy (priority lex))

See also

3.1.5.3 Conflict Resolution
defcontext
deftactic-lex
instantiation
conflict-set
fire-rule
-lex

Summary
Implements the \texttt{-lex} tactic.

Package
\texttt{kw}

Signature
\texttt{-lex instantiation1 instantiation2 => result}

Arguments
\texttt{instantiation1} \uparrow \text{An instantiation object.}
\texttt{instantiation2} \downarrow \text{An instantiation object.}

Values
\texttt{result} \quad \text{A boolean.}

Description
The conflict resolution tactic / function \texttt{-lex} returns true if and only if \texttt{instantiation1} is preferred to \texttt{instantiation2} by the conflict resolution tactic \texttt{-lex}, otherwise false. The function is intended to be used primarily by including it in the conflict resolution strategy for a context.

Examples

\begin{verbatim}
(defcontext my-context1 :strategy (-lex))
(defcontext my-context2 :strategy (priority -lex))
\end{verbatim}

See also

3.1.5.3 Conflict Resolution
defcontext
def tactic
c tactic
c l ex
instantiation
conflict-set
fire-rule
**list-all-inferencing-states**

*Function*

**Summary**
Returns a list of all the known inferencing states.

**Package**
`kw`

**Signature**
`list-all-inferencing-states => states`

**Values**
`states`  A list of inferencing states.

**Description**
The function `list-all-inferencing-states` returns a list of all the known inferencing states. Inferencing states become known when they are make and are known until they are destroyed.

**Examples**

```lisp
(list-all-inferencing-states)
```

**See also**
`destroy-inferencing-state`
`find-inferencing-state`
`*inferencing-state*`
`inferencing-state-name`
`make-inferencing-state`

**make-inferencing-state**

*Function*

**Summary**
Makes a new inferencing state.

**Package**
`kw`

**Signature**
`make-inferencing-state name &key set-current-p if-exists => state`
Arguments

name
Any object.

set-current-p
A boolean.

if-exists
Either :error, :supersede or :overwrite.

Values

state
An inferencing state.

Description

The function make-inferencing-state returns an inferencing state named by name.

If an inferencing state with the same name already exists (as compared using eql), then the value of if-exists determines what happens:

:error
A continuable error is signaled. Invoking the continue restart causes the existing inferencing state to be returned.

:supersede
The existing inferencing state is destroyed and a new one is returned.

:overwrite
The existing inferencing state is returned.

If set-current-p is non-nil, then *inferencing-state* is set to new inferencing state.

Examples

(make-inferencing-state 'my-state)

See also

destroy-inferencing-state
find-inferencing-state
*inferencing-state*
inferencing-state-name
list-all-inferencing-states

make-instance

Summary

Makes a CLOS or KnowledgeWorks structure object.

Package

common-lisp

Signature

make-instance class &rest initargs => object
Arguments

class ⇓ A class object or a symbol.
initargs ⇓ Initialization arguments for the object.

Values

object A new instance of class.

Description

The generic function make-instance makes a new instance of the class class.

If class is a CLOS class then the behavior is as specified by make-instance in the Common Lisp standard.

If class is a KnowledgeWorks structure class, then initargs are the same as those for the automatically defined constructor function of the structure.

The object is added to the object base of the current inferencing state.

Examples

(make-instance 'start)
(make-instance 'driver :location 'London :kb-name 'fred)

See also

def-kb-class
def-kb-struct
def-named-kb-class

mea

Conflict Resolution Tactic / Function

Summary

Implements the mea tactic.

Package

kw

Signature

mea instantiation1 instantiation2 => result

Arguments

instantiation1 ⇓ An instantiation object.
instantiation2 ⇓ An instantiation object.
Values

result  A boolean.

Description

The conflict resolution tactic / function \texttt{mea} returns true if and only if \texttt{instantiation1} is preferred to \texttt{instantiation2} by the conflict resolution tactic \texttt{mea}, otherwise false. The function is intended to be used primarily by including it in the conflict resolution strategy for a context.

Examples

\begin{verbatim}
(defcontext my-context1 :strategy (mea))
(defcontext my-context2 :strategy (priority mea))
\end{verbatim}

See also

3.1.5.3 Conflict Resolution
\texttt{defcontext}
\texttt{deftactic}
\texttt{~mea}
\texttt{instantiation}
\texttt{conflict-set}
\texttt{fire-rule}

\section*{-mea}
\textit{Conflict Resolution Tactic / Function}

Summary

Implements the \texttt{-mea} tactic.

Package

\texttt{kw}

Signature

\texttt{-mea instantiation1 instantiation2 => result}

Arguments

\begin{verbatim}
instantiation1 \downarrow
\text{An instantiation object.}
\\instantiation2 \downarrow
\text{An instantiation object.}
\end{verbatim}

Values

\begin{verbatim}
result \text{ A boolean.}
\end{verbatim}

Description

The conflict resolution tactic / function \texttt{-mea} returns true if and only if \texttt{instantiation1} is preferred to \texttt{instantiation2} by the conflict resolution tactic \texttt{-mea}, otherwise false. The function is intended to be used primarily by including it in the conflict
resolution strategy for a context.

Examples

(defcontext my-context1 :strategy (-mea))
(defcontext my-context2 :strategy (priority -mea))

See also

3.1.5.3 Conflict Resolution

defcontext
deftactic
mea
instantiation
conflict-set
fire-rule

named-kb-object

Class

Summary
A class that provides named objects.

Package
kw

Superclasses
standard-kb-object

Initargs

:kb-name The name of the object. The default is computed by calling gentemp with the name of the class.

Description
The class named-kb-object is the mixin class for named KnowledgeWorks CLOS objects.

Subclasses of named-kb-object are typically defined using the macro def-named-kb-class.

Examples

(defclass driver (named-kb-object)
  ((location) (allocated-truck)))

See also

get-kb-object
kb-name
def-named-kb-class
standard-kb-object
**no-debug**

*Function*

**Summary**

Turns debugging facilities off.

**Package**

kw

**Signature**

no-debug

**Description**

The function **no-debug** turns off all KnowledgeWorks debugging facilities. This means that rules and contexts cannot be single stepped or monitored, and no record is kept of when objects are created or modified. Execution speed of the rulebase is improved, and memory requirements reduced.

This should be called before compiling any rules or contexts that are to be optimized.

**Examples**

(no-debug)

**See also**

all-debug

---

**not**

*Backward Chaining Goal*

**Summary**

A goal that is satisfied when another goal fails.

**Package**

kw

**Signature**

not {condition}*

**Arguments**

condition

Any backward chaining goal.
Description

If the backward chaining goal `not` is used in a backward chaining clause, it succeeds if `condition` fails. In this usage, only one condition is allowed.

If `not` is used in a forward chaining pre-condition, it succeeds if any `condition` contained within it fail. In this usage, each `condition` may only contain expressions normally allowed in forward chaining pre-conditions (object base references and lisp tests). See 3.1.2 Forward Chaining Syntax for more details.

Examples

```lisp
(not (truck ?truck driver ?driver) (test ?driver))
```

See also

test

---

**order**

Conflict Resolution Tactic / Function

Summary

Implements the `order` tactic.

Package

kw

Signature

`order instantiation1 instantiation2 => result`

Arguments

instantiation1↓ An instantiation object.

instantiation2↓ An instantiation object.

Values

result A boolean.

Description

The conflict resolution tactic / function `order` returns true if and only if `instantiation1` is preferred to `instantiation2` by the conflict resolution tactic `order`, otherwise false. The function is intended to be used primarily by including it in the conflict resolution strategy for a context.

Examples

```lisp
(defcontext my-context1 :strategy (order))
(defcontext my-context2 :strategy (priority order))
```
See also

3.1.5.3 Conflict Resolution

defcontext
deftactic
-order
instantiation
conflict-set
fire-rule

-summary

Conflicts Resolution Tactic / Function

-implements the -order tactic.

-package

kw

-signature

-order instantiation1 instantiation2 => result

-arguments

instantiation1 An instantiation object.

instantiation2 An instantiation object.

-values

result A boolean.

-description

The conflict resolution tactic / function -order returns true if and only if instantiation1 is preferred to instantiation2 by the conflict resolution tactic -order, otherwise false. The function is intended to be used primarily by including it in the conflict resolution strategy for a context.

-examples

(defcontext my-context1 :strategy (-order))
(defcontext my-context2 :strategy (priority -order))

-see also

3.1.5.3 Conflict Resolution

defcontext
deftactic
-order
instantiation
conflict-set
Summary

Controls how much information is printed for an object.

Package

kw

Initial Value

nil

Description

The variable *print-verbose* controls how much information is printed for an object.

Normally objects in KnowledgeWorks are printed out in a brief form similar to ordinary CLOS objects. If *print-verbose* is set to t then all the slots and slot values are shown in its printed representation. Note that circularities cannot be detected.

Summary

Implements the priority tactic.

Package

kw

Signature

priority instantiation1 instantiation2 => result

Arguments

instantiation1
An instantiation object.

instantiation2
An instantiation object.

Values

result
A boolean.

Description

The conflict resolution tactic / function priority returns true if and only if instantiation1 is preferred to instantiation2 by the conflict resolution tactic priority, otherwise false. The function is intended to be used primarily by including it in the
conflict resolution strategy for a context.

Examples

(defcontext my-context1 :strategy (priority))
(defcontext my-context2 :strategy (recency priority))

See also

3.1.5.3 Conflict Resolution

defcontext
deftactic
-priority
instantiation
conflict-set
fire-rule

-priority

Conflict Resolution Tactic / Function

Summary

Implements the -priority tactic.

Package

kw

Signature

-priority instantiation1 instantiation2 => result

Arguments

instantiation1↓ An instantiation object.
instantiation2↓ An instantiation object.

Values

result A boolean.

Description

The conflict resolution tactic / function -priority returns true if and only if instantiation1 is preferred to instantiation2 by the conflict resolution tactic - -priority, otherwise false. The function is intended to be used primarily by including it in the conflict resolution strategy for a context.

Examples

(defcontext my-context1 :strategy (-priority))
(defcontext my-context2 :strategy (recency -priority))
Recency

Summary

Implements the recency tactic.

Package

kw

Signature

recency instantiation1 instantiation2 => result

Arguments

instantiation1 ↓ An instantiation object.
instantiation2 ↓ An instantiation object.

Values

result A boolean.

Description

The conflict resolution tactic / function recency returns true if and only if instantiation1 is preferred to instantiation2 by the conflict resolution tactic recency, otherwise false. The function is intended to be used primarily by including it in the conflict resolution strategy for a context.

Examples

(defcontext my-context1 :strategy (recency))
(defcontext my-context2 :strategy (priority recency))

See also

3.1.5.3 Conflict Resolution

defcontext
deftactic
defterm
priority
instantiation
conflict-set
fire-rule

Conflict Resolution Tactic / Function
**Summary**

Implements the `-recency` tactic.

**Package**

`kw`

**Signature**

`-recency instantiation1 instantiation2 => result`

**Arguments**

- `instantiation1`\[\rightarrow\] An instantiation object.
- `instantiation2`\[\rightarrow\] An instantiation object.

**Values**

- `result` A boolean.

**Description**

The conflict resolution tactic / function `-recency` returns true if and only if `instantiation1` is preferred to `instantiation2` by the conflict resolution tactic `-recency`, otherwise false. The function is intended to be used primarily by including it in the conflict resolution strategy for a context.

**Examples**

```
(defcontext my-context1 :strategy (recency))
(defcontext my-context2 :strategy (priority recency))
```

**See also**

3.1.5.3 Conflict Resolution

`context`
`defcontext`
`deftactic`
`recency`
`instantiation`
`conflict-set`
`fire-rule`
**reset**  
*Function*

**Summary**  
Clears all objects from the object base.

**Package**  
`kw`

**Signature**  
`reset`

**Description**  
The function `reset` clears all KnowledgeWorks objects (both KnowledgeWorks CLOS objects and KnowledgeWorks structures) from the object base of the current inferencing state.

The list of KnowledgeWorks classes remains unaffected.

**Examples**  

```
(reset)
```

**See also**  
`clear-all`  
`clear-rules`

---

**return**  
*Backward Chaining Goal*

**Summary**  
Removes the top-most context from the agenda.

**Package**  
`kw`

**Signature**  
$return$

**Description**  
The backward chaining goal `return` takes the topmost context on the agenda and makes it the current context, discarding the previous current context. When called from within a rule, rule execution continues to the end and the next rule to fire will be from the new current context.
**Examples**

*(return)*

**See also**

*context*

---

**\*signal-kb-name-clash\***

**Variable**

**Summary**

Controls the behavior if name clashes occur in object creation.

**Package**

*kw*

**Initial Value**

*:error*

**Description**

The variable *\*signal-kb-name-clash\* determines behavior when creating a new named KB object with the same name as an existing KB object.

The possible values are:

*:error* Signals an error; continuing will replace the old object with the new object.

*:warn* Signals a warning and replaces the old object with the new object.

*:quiet* Replaces the old object with the new object.

---

**specificity**

**Conflict Resolution Tactic / Function**

**Summary**

Implements the specificity tactic.

**Package**

*kw*

**Signature**

**specificity instantiation1 instantiation2 => result**

**Arguments**

`instantiation1` An instantiation object.
Instantiation2

An instantiation object.

Values

result A boolean.

Description

The conflict resolution tactic / function specificity returns true if and only if instantiation1 is preferred to instantiation2 by the conflict resolution tactic specificity, otherwise false. The function is intended to be used primarily by including it in the conflict resolution strategy for a context.

Examples

(defcontext my-context1 :strategy (specificity))
(defcontext my-context2
  :strategy (priority specificity))

See also

3.1.5.3 Conflict Resolution
defcontext
deftactic
-specificity
instantiation
conflict-set
fire-rule

-specificity

Conflict Resolution Tactic / Function

Summary

Implements the specificity tactic.

Package

kw

Signature

-specificity instantiation1 instantiation2 => result

Arguments

instantiation1 An instantiation object.
instantiation2 An instantiation object.

Values

result A boolean.
Description
The conflict resolution tactic / function \(-\text{specificity}\) returns true if and only if \(\text{instantiation1}\) is preferred to \(\text{instantiation2}\) by the conflict resolution tactic \(-\text{specificity}\), otherwise false. The function is intended to be used primarily by including it in the conflict resolution strategy for a context.

Examples

```lisp
(defun my-context1 :strategy (-specificity))
(defun my-context2 :strategy (priority -specificity))
```

See also

3.1.5.3 Conflict Resolution
defcontext
deftactic

**standard-context**

Summary
The standard meta-interpreter context.

Package
kw

Signature

standard-context

Description
The backward chaining goal \texttt{standard-context} is the built-in goal that implements a meta-interpreter for the default (normal) behavior of a context. It is as if defined by the rule:

```lisp
(defrule standard-context :backward
    ((standard-context)
     <--
     (start-cycle)
     (instantiation ?instantiation)
     (fire-rule ?instantiation)
     (cut)
     (standard-context)))
```

Examples

```lisp
(defun my-context1 :meta (((format t "-%Entering context MY-CONTEXT1")))
```
(standard-context))

See also
defcontext
start-cycle
instantiation
fire-rule

standard-kb-object

Class

Summary
A class of objects for use in the object base.

Package
kw

Superclasses
standard-object

Description
The class standard-kb-object is the mixin class for (unnamed) KnowledgeWorks CLOS objects.

Subclasses of standard-kb-object are typically defined using the macro def-kb-class.

Examples

(defclass driver (standard-kb-object)
  ((location) (allocated-truck)))

See also
def-kb-class
named-kb-object

start-cycle

Backward Chaining Goal

Summary
Used in the meta-interpreter to start the cycle.

Package
kw
7 Reference Guide

Signature

start-cycle

Description

The backward chaining goal start-cycle is only relevant when writing a meta-interpreter for a context. start-cycle must be called at the start of every forward chaining cycle as it performs some essential housekeeping.

Examples

(start-cycle)

See also

fire-rule
instantiation
defcontext
standard-context

start-kw

Summary

Starts the KnowledgeWorks programming environment.

Package

kw

Signature

start-kw &key host

Arguments

host \(\downarrow\)
A string.

Description

The function start-kw starts the KnowledgeWorks programming environment from the initial prompt when the KnowledgeWorks image is started. If the LispWorks IDE is already running, start-kw adds the KnowledgeWorks menu so that the podium becomes the KnowledgeWorks Podium.

On GTK+ and Motif the environment is displayed on the machine specified by host, defaulting to the machine on which the KnowledgeWorks image is running. Other platforms ignore host.

Examples

(start-kw)

(start-kw :hostname "display1:0")
**test**  
*Backward Chaining Goal*

**Summary**
Evaluates a Lisp form as a backward chaining goal.

**Package**
kw

**Signature**
`test lisp-form`

**Arguments**
- `lisp-form`  
  A single Lisp form.

**Description**
The backward chaining goal `test` succeeds if and only if `lisp-form` returns a non-nil value. Any currently bound variables may be used in the lisp form.

`test` can also be used as a forward chaining pre-condition, as described in [3.1.2 Forward Chaining Syntax](#).

**Examples**
```
(test (> ?c 10))
(test (not (and (eq ?a ?b) (member ?b ?c))))
```

**undefcontext**  
*Macro*

**Summary**
Removes a named context and its rules.

**Package**
kw

**Signature**
`undefcontext context-name &rest ignore`

**Arguments**
- `context-name`  
  A symbol which names a context.
- `ignore`  
  Ignored arguments.
Description

The macro **undefcontext** removes the context named *context-name* and all the rules in it.

*ignore* is not used and is only provided so that "un" may be prepended to a context definition in an editor buffer and evaluated to remove the context.

Examples

```lisp
(undefcontext my-context)
```

See also

**defcontext**

---

**undefrule**

Macro

Summary

Removes a rule.

Package

**kw**

Signature

**undefrule** **rule-name** &rest **ignore**

Arguments

- **rule-name**: A symbol which names a rule.
- **ignore**: Ignored arguments.

Description

The macro **undefrule** removes the rule named *rule-name* and any unfired instantiations of that rule.

*ignore* is not used and is only provided so that "un" may be prepended to a rule definition in an editor buffer and evaluated to remove the rule.

Examples

```lisp
(undefrule my-rule1)
```

See also

**defrule**
with-rule-actions

Summary
Allows rule syntax to be embedded in Lisp code.

Package
kw

Signature

\texttt{with-rule-actions} \texttt{bound-variables} \&\texttt{body} \texttt{body} \Rightarrow \texttt{successp}

Arguments

\texttt{bound-variables} \downarrow \ A \text{list of variables (each starting with \texttt{?})}.

\texttt{body} \downarrow \ A \text{rule body}.

Values

\texttt{successp} \downarrow \ A \text{boolean}.

Description
The macro \texttt{with-rule-actions} \text{macro enables rule syntax to be embedded within Lisp.}

\texttt{body} \text{is executed just as if it were the right hand side of a forward or backward chaining rule. All variables in} \texttt{body} \text{(each starting with \texttt{?}) are taken to be unbound unless found in the list} \texttt{bound-variables}, \text{in which case its value is taken from the Lisp variable of the same name.} \texttt{with-rule-actions} \text{is similar to the function} \texttt{any} \text{but can be compiled for efficiency.}

\texttt{successp} \text{is} \texttt{t} \text{if the body succeeds (that is, all clauses are successfully executed) or} \texttt{nil} \text{if any of the clauses fail.}

Any subgoals that match the object base will only find objects from the current inferencing state.

Examples

\begin{verbatim}
(defun my-fn (?x)
  "prints all the lists which append to give ?x and then returns NIL"
  (with-rule-actions (?x)
    (append ?a ?b ?x)
    ((format t "~-S and ~S append to give ~S"
             ?a ?b ?x))
    (fail)))
\end{verbatim}

See also

\texttt{any}
Appendix A: Common Prolog

A.1 Introduction

A.1.1 Overview

Common Prolog is a logic programming system within Common Lisp. It conforms closely to Edinburgh Prolog and at the same time integrates well with Lisp. The basic syntax of Common Prolog is Lisp-like, but an Edinburgh syntax translator is included that provides the ability to use pre-existing code. The implementation of Common Prolog was motivated by the desire to use the logic programming paradigm without having to give up the advantages of a Lisp development environment. Common Prolog is tightly integrated with Lisp and can be easily used in a mixed fashion with Lisp definitions even within the same source file. Common Prolog predicates are compiled into Lisp functions which may then be compiled by a standard Lisp compiler. Substantial effort has gone into providing a powerful debugging environment for Common Prolog, so that it can be used when building serious applications. The implementation of Common Prolog is based loosely on the Warren Abstract Machine (WAM) modified to take advantage of a Lisp environment's built-in support for control flow and memory allocation. (For more details of the WAM, see An Abstract Prolog Instruction Set, by David H D Warren, Technical Note 309, SRI International, October 1983.)

A.1.1.1 Starting Common Prolog

Common Prolog may be loaded into an image with the function call:

```
(require "prolog")
```

This will load the Common Prolog system. If Common Prolog will be used extensively, it may be worthwhile to save an image with it pre-loaded. Alternatively, you may simply insert the call above into your LispWorks initialization file (usually .lispworks).

For information about saving an image and the LispWorks initialization file, see the Release Notes and Installation Guide.

Note: If you load KnowledgeWorks, then Common Prolog is loaded as part of this.

A.2 Syntax

Common Prolog uses a Lisp-like syntax in which variables are prefixed with "?" and normal Lisp prefix notation is used. Goals are represented as either lists or simple vectors e.g. (reverse (1 2 3) ?x) or #(member ?x (1 2 3)). A symbol beginning with ? may be escaped by prefixing another ?.i.e. ?foo is the variable named foo; ??foo is the symbol ?foo.

The definition of append/3 from Prolog:

```
append([], X, X).
append([U|X], Y, [U|Z]) :-
    append(X, Y, Z)
```

translates to:
Unlike many Lisp-based logic systems, Common Prolog uses simple vectors to represent Prolog structured terms. Thus, functor, arg, and =.. all behave in a standard fashion:

```
(arg 2 (foo 3 4) (3 4))
(arg 2 #(foo 3 4) 4)
(functor (foo 3 4) \ 2)
(functor #(foo 3 4) foo 2)
(=.. #(foo 3 4) (foo 3 4))
(=.. (foo 3 4) (\ foo (3 4)))
```

### A.3 Defining Relations

The normal method of defining relations in Common Prolog is to use the `defrel` macro:

```
(defrel <relation name>
  [(declare declaration*)]
  <clause1>
  ...
  <clauseN>)
```

where each `<clause>` is of the form:

```
(<clause-head>
 <subgoal1>
 ...
 <subgoalN>)
```

and declarations may include: `(mode arg-mode*)` and any of the normal Lisp optimization declarations. Mode declarations determine how much clause indexing will be done on the predicate and can also streamline generated code for a predicate that will only be used in certain ways. A mode declaration consists of the word "MODE" followed by a mode spec for each argument position of the predicate. The possible argument mode specs are:

- `?` Generate completely general code for this arg and don't index on it.
- `?*` Generate completely general code and index.
- `+` Generate code assuming this argument will be bound on entry and index.
- `-` Generate code assuming this argument will be unbound on entry and don't index.

The default mode specs are `?*` for the first argument and `?` for all the rest.

### A.4 Using The Logic Interpreter

The Common Prolog system comes with a built-in `read-query-print` loop similar to a Prolog interpreter loop. To run it, make sure the common-prolog package is accessible and type: `(rqp)`. You will be presented with the prompt: `==>`. At this point you may type in goal expressions, for example:

```
|==> (append ?x ?y (1 2))
```
Now Common Prolog is waiting for you to indicate whether or not you wish more solutions. If you press Return, you will get the message OK and return to the top level:

```prolog
?X = NIL
?Y = (1 2)<RETURN>
OK.
```

A.4.1 Multiple Solutions

If you hit ; (semicolon) following the retrieval of a solution, the system will attempt to resatisfy your goal:

```
?X = NIL
?Y = (1 2);
?X = (1)
?Y = (2);
?X = (1 2)
?Y = NIL;
NO.
```

When no more solutions remain, NO. is displayed and you are back at the top level.

A.4.2 Multiple Goals

To request the solution of multiple goals, use: (and <goal1> ... <goalN>).

For example:

```
=> (and (member ?x (2 3)) (append (?x) (foo) ?y))
X = 2
Y = (2 FOO)
OK.
```

A.4.3 Definitions

It is possible to type logic definitions directly into the interpreter. The resulting Lisp code will be compiled in memory and you may use the definition immediately, for example:

```
=> (defrel color
    ((color red))
    ((color blue))
    ((color green)))
... various compilation messages ...
```
Appendix A: Common Prolog

A.4.4 Exiting the Interpreter

The Common Prolog interpreter may be exited by typing:

```prolog
==> (halt)

A.5 Accessing Lisp From Common Prolog

It is apparent from the Common Prolog syntax that the first element of any valid goal expression must be a symbol. Common Prolog takes advantage of this fact and gives a special interpretation to a goal with a list in the first position. A list in the `car` of a goal is treated as a Lisp expression with normal Lisp evaluation rules. Any logic variables in the expression are instantiated with their values. (They must be bound). The rest of the goal expression should be a list of expressions to be unified with the values returned by the Lisp evaluation. Any extra values returned are ignored, and any extra expressions in the tail of a goal are unified with new unbound variables.

A.5.1 Examples

```prolog
==> ((print "foo"))
"foo"
YES.

==> (and (= ?x 3) ((* ?x ?x) ?y))
   ; Note that "?y" is unified with 9

?X = 3
?Y = 9

==> ((* 3 3) 10)

NO.

==> ((floor 3 4) ?x ?y)
?X = 0
?Y = 3

==> ((floor 3 4) ?x)
?X = 0

==> ((* 3 4) ?x ?y)
?X = 12
?Y = ?0
   ; note that system generated variables look like:
   ; ?<integer>

```
A.6 Calling Prolog From Lisp

There are several entry points provided for calling Prolog from Lisp. The main interface function is called `logic` and has numerous options. The basic form is:

```lisp
(logic <goal>
   :return-type <return-type>
   :all <all-type>
   :bag-exp <bag-exp>)
```

The keyword arguments are interpreted as follows:

- **:return-type** describes what to do with a solution when one is found. Possible values of **:return-type** are:
  - **:display** Display variable bindings and prompt user (the option used by the `read-query-print` loop).
  - **:fill** Instantiate the goal expression and return it.
  - **:bag** Instantiate `<bag-exp>` and return it.
  - **:alist** Return an alist of variables and bindings.

The default is **:fill**.

- **:all** tells what to do with multiple solutions. Possible values of **:all** are:
nil Return the first solution.

:values Return multiple solutions as multiple values.

:list Return a list of the solutions.

:bag-exp is an expression that should be instantiated with the bindings from a solution. This is only meaningful if :return-type is :bag.

A.6.1 Examples

(logic '(color ?x) :return-type :display)

writes:

?X = RED<wait for input>

(logic '(color ?x) :return-type :fill)

returns:

(COLOR RED)
T

(logic '(color ?x) :return-type :alist)

returns:

((?X . RED))
T

(logic '(color ?x) :all :list)

returns:

((COLOR RED) (COLOR BLUE) (COLOR GREEN))
T

(logic '(color ?x)
    :return-type :bag
    :bag-exp '(:?x is a color)
    :all :values)

returns:

(RED IS A COLOR)
(BLUE IS A COLOR)
(GREEN IS A COLOR)
A.6.2 Interface Functions

There are three additional ways to call logic, which are described in this section.

A.6.2.1 any, findall and findallset

Three simple interface functions call logic. They are any, findall, and findallset. Each takes two arguments: a result expression to instantiate and a goal expression. any returns the first solution found. findall returns all solutions. findallset returns all solutions deleting duplicates.

Assuming the definitions for fact and color from the previous examples.

| (any '(?x is the factorial of 5) '(fact 5 ?x)) |

returns:

| (120 IS THE FACTORIAL OF 5) |

| (findall '(?x is a color) '(color ?x)) |

returns:

| ((RED IS A COLOR) (BLUE IS A COLOR) |

| (GREEN IS A COLOR)) |

| (findall '(?y (or (= ?y 5) (= ?y 5))) |

returns:

| (5 5) |

| (findallset '(?y (or (= ?y 5) (= ?y 5))) |

returns:

| (5) |

findall and findallset will hang if a goal expression generates an infinite solution set.

More powerful all solution predicates (bagof and setof) are available from within Common Prolog.

A.6.2.2 deflogfun

A different interface is available for predicates which will be called often from Lisp. The macro deflogfun may be used to generate normal Lisp functions that run with precompiled goals.

(deflogfun break-up (y) (append ?a ?b y) (?a ?b))

then:
Appendix A: Common Prolog

```lisp
(break-up '(foo bar baz))
```

returns:

```lisp
(NIL (FOO BAR BAZ))
T
```

```lisp
(break-up '(foo bar baz) :all :values)
```

returns:

```lisp
(NIL (FOO BAR BAZ))
((FOO) (BAR BAZ))
((FOO BAR) (BAZ))
((FOO BAR BAZ) NIL)
```

```lisp
(break-up '(foo bar baz) :all :list)
```

returns:

```lisp
((NIL (FOO BAR BAZ))
 ((FOO) (BAR BAZ))
 ((FOO BAR) (BAZ))
 ((FOO BAR BAZ) NIL))
T
```

The generated function works like the Lisp functions `any` and `findall`, returning solutions to a prolog expression.

The form:

```lisp
(deflogfun name args sample-expr return-expr)
```

defines a Lisp function called `name`, whose lambda list is the list `args`. The function will also take a keyword argument :all. If the function is called with :all nil (the default), then it returns the first solution, like `any`. If the function is called with :all t, then it returns a list of all the solutions, like `findall`. If the function is called with :all :values, then it returns multiple values, with one value per solution.

The `sample-expr` is like the second argument to `any`, that is, it is the prolog query expression. The `return-expr` is like the first argument to `clog:any`, that is, it defines how the result will be formed from the results of the query. If any of the symbols mentioned in `args` appears in `sample-expr` or `return-expr`, then its value is substituted. All other symbols in `sample-expr` and `return-expr` remain unchanged.

### A.6.2.3 with-prolog

A final interface mechanism is `with-prolog`, which allows you to embed prolog into an arbitrary lisp function. Lisp variables are referenced in Prolog using "?.<name>".

```lisp
(defun palindromep (x)
  (with-prolog
    (append ?a (?b . ?c) ?.x) ; note ".x" reference
    (or (reverse ?a ?c)
      (reverse ?a (?b . ?c))))))

(palindromep '(yes no maybe))
```

returns:

A
NIL

(palindromep 'yes no maybe no yes)

returns:

T

The body of a with-prolog returns t if it succeeds and a non-local exit is not executed. It returns nil on failure.

A.7 Debugging

Common Prolog provides a standard 4-port debugging model (call exit redo fail).

Tracing, Spy Points, Leashing, and Interactive Debugging are each discussed separately in this section.

A.7.1 Tracing

Exhaustive tracing is available with Common Prolog through the use of: (trace). After executing (trace), all goals will be displayed until control is returned to the top level loop, nodebug is executed or notrace is executed.

A.7.1.1 Tracing rules

You can turn on tracing for backward chaining from Lisp by running:

(clog:logic '(and (clog:unleash) (trace)))

There are no command line tools for tracing forward chaining rules directly, but the RHS of each rule is run using a backward chaining rule with the same name, so they also appear when you trace backward chaining.

You could also add tracing to forward rules by defining a Meta Rule Protocol, for example like the explanation facility described in 6.1.1.3 A Simple Explanation Facility.

A.7.2 Spy Points

Spy points are the most important debugging facility in Common Prolog. They are used in the same way trace is used in Lisp. After executing (spy foo), all events associated with satisfying foo goals will be traced and the user will enter a debugging command loop at every port (see A.7.4 Interactive Debugging below). A user can also specify (spy (foo 3)), (spy (foo bar)), or (spy ((foo 3) bar)) to place spy points on foo goals with arity 3, on all predicates for foo and bar, or on foo with arity 3 and all predicates for bar respectively. Spy points are turned off with (nospy <spypoints>). If no spy points are mentioned, nospy will turn off all spy points.

A.7.3 Leashing

Leashing allows the user to control execution while tracing for goals that are not spied. Spied goals cause execution to enter a debugging command loop whenever they are reached. Leashing provides the same functionality for unspied goals. A user may choose to enter a debugging command loop at any subset of ports by using (leash events) where events may be: call, redo, exit or fail. Leashing may be turned off using (unleash).
A.7.4 Interactive Debugging

When Common Prolog execution enters a debugging command loop, the user has many options, which may be listed with ?., for example:

```
===> (spy member)

((MEMBER 2))
YES.
OK.

===> (member 3 ?x)

[1] CALL: (MEMBER 3 ?0)? ? <- user types ?

(c)reep     - turn on exhaustive tracing
(s)kip      - skip until another port is
              reached for this goal
(l)eap      - turn off tracing until a spy
              point or this goal is reached
(b)reak     - enter a recursive
              read/query/print loop
(d)isplay   - display a listing for the
              current goal
(q)uit      - quit to top level
(r)etry     - try to satisfy this goal again
(f)ail      - cause the current goal to fail
(a)abort     - exit Common Prolog
?           - display this information

In a little more detail...

creep   - causes exhaustive tracing of the
          next goal
skip    - ignores spy points and executes
          without displaying anything until
          this goal is reached again
          either at an exit, fail,
          or redo port
leap    - turns off exhaustive tracing until
          a spy point or this goal is
          reached
break   - enters a recursive interpreter loop
          so that the user may query
          values, redefine a predicate, etc.
display - uses "listing" to display the
          listing of the current goal
quit    - returns to the top level interpreter
          loop
retry   - causes execution to return to the
          call port of this goal as if
          this goal had just been reached for
          the first time.
fail    - causes execution to jump to the fail
          port of this goal
abort   - completely exit Common Prolog
```

Continuing the example:

```
d <- user selects display

Compiled procedure:

119
```
Appendix A: Common Prolog

```

```
(DEFREL MEMBER
   ((MEMBER ?X (?X . ?)))
   ...

[1] EXIT: (MEMBER 3 (3 . ?))? r
   ...

[1] CALL: (MEMBER 3 ?0)? f <-- user selects fail

[1] FAIL: (MEMBER 3 ?0)? r <-- one more time

[1] CALL: (MEMBER 3 ?0)? s <-- skip

[1] EXIT: (MEMBER 3 (3 . ?))? l <-- leap

?X = (3 . ?0); <-- more solutions

[1] REDO: (MEMBER 3 (3 . ?))? c <-- creep

[2] CALL: (MEMBER 3 ?0)? b <-- break

==> (nospy)
NIL <-- current spylist
YES.
OK.

==> (halt) <-- return to original execution
? l <-- leap

?X = (?0 3 . ?1)<cr>

OK.

Another example:

==> (defrel reverse
   ((reverse () ()))
   ((reverse (?x . ?y) ?z)
    (reverse ?y ?w)
    (append ?w (?x) ?z)))
<noise..>

?X = ?0
?Y = ?1
?Z = ?2
?W = ?3

OK.

==> (defrel append
   ((append () ?x ?x))
   ((append (?u . ?x) ?y (?u . ?z))
    (append ?x ?y ?z)))
<noise..>

?X = ?0
?U = ?1
?Y = ?2
```
Appendix A: Common Prolog

?- Z = 3
OK.

==> (unleash)
YES.
OK.

==> (trace)
YES.
OK.

==> (reverse (1 2 3) ?x)

[1] CALL: (REVERSE (1 2 3) ?0)
[2] CALL: (REVERSE (2 3) ?0)
[3] CALL: (REVERSE (3) ?0)
[4] CALL: (REVERSE NIL ?0)
[4] EXIT: (REVERSE NIL NIL)
[5] CALL: (APPEND NIL (3) ?0)
[5] EXIT: (APPEND NIL (3) (3))
[3] EXIT: (REVERSE (3) (3))
[6] CALL: (APPEND (3) (2) ?0)
[7] CALL: (APPEND NIL (2) ?0)
[7] EXIT: (APPEND NIL (2) (2))
[6] EXIT: (APPEND (3) (2) (3 2))
[2] EXIT: (REVERSE (2 3) (3 2))
[8] CALL: (APPEND (3 2) (1) ?0)
[9] CALL: (APPEND (2) (1) ?0)
[10] CALL: (APPEND NIL (1) ?0)
[10] EXIT: (APPEND NIL (1) (1))
[9] EXIT: (APPEND (2) (1) (2 1))
[8] EXIT: (APPEND (3 2) (1) (3 2 1))
[1] EXIT: (REVERSE (1 2 3) (3 2 1))
?X = (3 2 1);

[1] REDO: (REVERSE (1 2 3) (3 2 1))
[8] REDO: (APPEND (3 2) (1) (3 2 1))
[9] REDO: (APPEND (2) (1) (2 1))
[10] REDO: (APPEND NIL (1) (1))
[10] FAIL: (APPEND NIL (1) ?0)
[9] FAIL: (APPEND (2) (1) ?0)
[8] FAIL: (APPEND (3 2) (1) ?0)
[2] REDO: (REVERSE (2 3) (3 2))
[6] REDO: (APPEND (3) (2) (3 2))
[7] REDO: (APPEND NIL (2) (2))
[7] FAIL: (APPEND NIL (2) ?0)
[6] FAIL: (APPEND (3) (2) ?0)
[3] REDO: (REVERSE (3) (3))
[5] REDO: (APPEND NIL (3) (3))
[5] FAIL: (APPEND NIL (3) ?0)
[4] REDO: (REVERSE NIL NIL)
[4] FAIL: (REVERSE ?0)
[3] FAIL: (REVERSE (3) ?0)
[2] FAIL: (REVERSE (2 3) ?0)
[1] FAIL: (REVERSE (1 2 3) ?0)
NO.
A.8 Common Prolog Macros

Macros may be defined within the logic system using the form:

\[
\text{(defrelmacro } \text{name} \text{ <arg-list> <body>)}
\]

which is effectively the same as a Common Lisp \text{defmacro}. Logic macros are expanded before variable translation so that logic variables may be treated as atoms. \text{defrelmacro} forms must have a fixed number of arguments. This allows different predicates with the same name but different arities to be defined. If you want to define a special form with an arbitrary number of arguments, use \text{defrel-special-form-macro}.

A.8.1 Example

\[
\text{(defrelmacro append3 } \text{x y z w)}
\]

\[
\text{(let } ((\text{iv} (\text{make-internal-var})))
\]

\[
\text{\indent} (\text{and} \text{ (append x y iv})
\]

\[
\text{\indent} (\text{append iv z w}))
\]

\[
==\Rightarrow (\text{append3 } \text{1} \text{2} \text{3} \text{?y})
\]

\[
?Y = (1 2 3)
\]

A.9 Defining Definite Clause Grammars

The \text{defgrammar} macro can be used to define a definite clause grammar (DCG), which is a relation that determines whether the start of a list of tokens (a \text{sentence}) matches a particular grammar. The remaining tokens in the list become the \text{sentence tail}.

The relation has the form:

\[
(\text{<grammar name> <sentence> <sentence tail> <extra argument>*})
\]

where the \text{<extra argument>} items are terms defined below.

The syntax of the \text{defgrammar} macro is:

\[
(\text{defgrammar } \text{<grammar name>}
\]

\[
\text{<rule>*)}
\]

\[
<\text{rule}> ::= (<\text{lhs}> <\text{rhs}>*)
\]

\[
<\text{lhs}> ::= <\text{grammar name}>
\]

\[
\text{\mid} (<\text{grammar name} <\text{term}>*)
\]

\[
\text{\mid} ((<\text{grammar name} <\text{term}>* <\text{newterm}>*))
\]

\[
<\text{rhs}> ::= <\text{atom}>
\]

\[
\text{\mid} <\text{var}>
\]

\[
\text{\mid} (<\text{other grammar name} <\text{term}>*)
\]

\[
\text{\mid} <\text{lisp clause}>
\]

\[
\text{\mid} (<\text{call} <\text{term}>)
\]

\[
\text{\mid} (<\text{cut})
\]

\[
<\text{lisp clause}> ::= (<\text{non-atomic lisp form} <\text{term}>*)
\]

\[
<\text{non-atomic lisp form}> ::= (<\text{lisp function name} <\text{lisp arg}>*)
\]

Appendix A: Common Prolog

A.8 Common Prolog Macros

Macros may be defined within the logic system using the form:

\[
(\text{defrelmacro } \text{name} \text{ <arg-list> <body>})
\]

which is effectively the same as a Common Lisp \text{defmacro}. Logic macros are expanded before variable translation so that logic variables may be treated as atoms. \text{defrelmacro} forms must have a fixed number of arguments. This allows different predicates with the same name but different arities to be defined. If you want to define a special form with an arbitrary number of arguments, use \text{defrel-special-form-macro}.

A.8.1 Example

\[
(\text{defrelmacro append3 } \text{x y z w})
\]

\[
(\text{let } ((\text{iv} (\text{make-internal-var})))
\]

\[
\text{\indent} (\text{and} \text{ (append x y iv})
\]

\[
\text{\indent} (\text{append iv z w}))
\]

\[
==\Rightarrow (\text{append3 } \text{1} \text{2} \text{3} \text{?y})
\]

\[
?Y = (1 2 3)
\]

A.9 Defining Definite Clause Grammars

The \text{defgrammar} macro can be used to define a definite clause grammar (DCG), which is a relation that determines whether the start of a list of tokens (a \text{sentence}) matches a particular grammar. The remaining tokens in the list become the \text{sentence tail}.

The relation has the form:

\[
(\text{<grammar name> <sentence> <sentence tail> <extra argument>*})
\]

where the \text{<extra argument>} items are terms defined below.

The syntax of the \text{defgrammar} macro is:

\[
(\text{defgrammar } \text{<grammar name>}
\]

\[
\text{<rule>*})
\]

\[
<\text{rule}> ::= (<\text{lhs}> <\text{rhs}>*)
\]

\[
<\text{lhs}> ::= <\text{grammar name}>
\]

\[
\text{\mid} (<\text{grammar name} <\text{term}>*)
\]

\[
\text{\mid} ((<\text{grammar name} <\text{term}>* <\text{newterm}>*))
\]

\[
<\text{rhs}> ::= <\text{atom}>
\]

\[
\text{\mid} <\text{var}>
\]

\[
\text{\mid} (<\text{other grammar name} <\text{term}>*)
\]

\[
\text{\mid} <\text{lisp clause}>
\]

\[
\text{\mid} (<\text{call} <\text{term}>)
\]

\[
\text{\mid} (<\text{cut})
\]

\[
<\text{lisp clause}> ::= (<\text{non-atomic lisp form} <\text{term}>*)
\]

\[
<\text{non-atomic lisp form}> ::= (<\text{lisp function name} <\text{lisp arg}>*)
\]
Appendix A: Common Prolog

<grammar name> is the same symbol as the one naming the defgrammar.

<other grammar name> is a symbol naming another defgrammar.

<atom> is an atom, which forms the words of the sentence to be matched.

<var> is a variable reference.

<term> and <newterm> are any Common Prolog logic expression, including a variable.

<lisp function name> is a symbol naming a Lisp function.

<lisp arg> is any Lisp form, which is evaluated and passed to the function.

Within the <lhs>, extra arguments can be added by specifying <term>s. Every <rule> must specify the same <grammar name> as the defgrammar form and have the same number of extra arguments.

If the <lhs> specifies <newterm> forms, then they are pushed onto start of the sentence tail if the rule matches. This corresponds to the "pushback lists" or "right-hand context" in traditional Prolog DCG syntax.

The meaning of the various <rhs> items is as follows:

- <atom> matches that atom in the sentence.
- <var> is unified with the next item in the sentence.
- (other grammar name <term>* ) calls the grammar relation other grammar name on the rest of the sentence. The optional <term> arguments are passed to the relation as its extra arguments.
- <lisp clause> evaluates the <non-atomic lisp form> as a Lisp form and unifies the values that it returns with the <term>s that follow it.
- (call <term>) calls <term> as a normal Prolog relation.
- (cut) calls the normal Prolog cut relation.

The phrase predicate can be used to call a DCG.

A.9.1 Examples

Here are some examples of using defgrammar.

A.9.1.1 Example 1: A simple definition.

This example shows the Common Prolog translation of the grammar shown at the top of http://cs.union.edu/~striegnk/learn-prolog-now/html/node59.html.

(defgrammar gram-det
  (gram-det the)
  (gram-det a))

(defgrammar gram-n
  (gram-n woman)
  (gram-n man))

(defgrammar gram-v
  (gram-v shoots))

(defgrammar gram-np
  (gram-np (gram-det) (gram-n)))

123
A.9.1.2 Example 2: Using extra arguments.

```prolog
(defgrammar one-of
  ((one-of ?word) ?word))

(defgrammar two-of
  ((two-of ?word) (one-of ?word) (one-of ?word)))
```

Each of these defines a 3-ary relation, whose extra argument is the word to match. When the relations are called, the word will typically be bound to a symbol from the sentence to match.

The following will succeed and bind ?x to the list (foo bar):

```prolog
(clog:any '?x ' (gram-s (a woman shoots foo bar) ?x))
(clog:any '?x ' (gram-s (a woman shoots the man foo bar) ?x))
```

A.10 Edinburgh Syntax

Common Prolog provides a translator from Edinburgh syntax to allow users to port pre-existing code.

The `consult` predicate operates only on `.pl` files:

- `consult('xxx.pl')` means consult file `xxx.pl`.
- `consult('xxx')` means find a file named `xxx.pl` and consult it.

The `reconsult` predicate can operate on a Lisp source file, since `compile_and_reconsult('xxx.pl')` produces a Lisp binary file `xxx.?fasl`. That is, `reconsult` will load fasl and lisp files as well as `.pl` files:

- `reconsult('xxx.pl')` means consult file `xxx.pl`.
- `reconsult('xxx')` means look for a file named `xxx.?fasl` and load it, or if none found, look for `xxx.pl` and consult it, or if none found look for `xxx.lisp` and load it, or load `xxx`.

Loading a compiled file is equivalent to `reconsult`.

`compile_and_reconsult` compiles a file and reconsults the result.
Edinburgh syntax may also be used to interact with Common Prolog through the use of a different read-query-print loop. To use Edinburgh syntax, use (erqp) instead of (rqp) to start your command loop.

### A.11 Graphic Development Environment

Common Prolog includes a graphic environment, consisting of a specialized listener and graphic debugging tools. With the debugging tools it is possible to step through a program at the source level and control the 4-port debugger using the mouse. Call trees for predicates may also be displayed and manipulated.

The specialized listener provides mouse control over:

- File editing, compiling, consulting and reconsulting.
- Debugging control flow (creep, leap, skip, etc.).
- Leashing of debugging ports.
- The addition and deletion of spy points.

The Logic Listener interaction is similar to a normal Lisp Listener and will accept normal Lisp expressions except that:

1. Any expression that can be interpreted as Common Prolog will be handled by the Logic subsystem.
2. If a line consisting of just `?-` is entered, the Logic Listener will go into an Edinburgh (erqp) loop.

### A.12 Built-in Predicates

The built-in predicates listed in the table below are exported from the common-prolog (clog) package.

<table>
<thead>
<tr>
<th>Predicate</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>/== (?x ?y)</td>
<td>same as Prolog ==</td>
</tr>
<tr>
<td>= (?x ?y)</td>
<td>standard Prolog</td>
</tr>
<tr>
<td>=.. (?x ?y)</td>
<td>standard Prolog</td>
</tr>
<tr>
<td>== (?x ?y)</td>
<td>standard Prolog</td>
</tr>
<tr>
<td>&lt;= (?x ?y)</td>
<td>same as Prolog except all variables sort as identical</td>
</tr>
<tr>
<td>&lt;=&lt; (?x ?y)</td>
<td>ditto</td>
</tr>
<tr>
<td>&gt;= (?x ?y)</td>
<td>ditto</td>
</tr>
<tr>
<td>@&gt; (?x ?y)</td>
<td>ditto</td>
</tr>
<tr>
<td>@&gt;= (?x ?y)</td>
<td>ditto</td>
</tr>
<tr>
<td>append (?x ?y ?z)</td>
<td>standard Prolog</td>
</tr>
<tr>
<td>arg (+index +term ?value)</td>
<td>standard Prolog</td>
</tr>
<tr>
<td>asserta (+exp)</td>
<td>standard Prolog</td>
</tr>
<tr>
<td>assertz (+exp)</td>
<td>standard Prolog</td>
</tr>
<tr>
<td>atomic (?x)</td>
<td>standard Prolog</td>
</tr>
</tbody>
</table>
| bagof (+goal . +ex-vars) (?bag) | standard Prolog (unusual syntax) (
<p>| call (+exp)    | standard Prolog                                  |
| clause (+head ?tail) | standard Prolog                                |</p>
<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>debug()</code></td>
<td>Cause debugging information to be saved for each call whether it is spied or not</td>
</tr>
<tr>
<td><code>debugging()</code></td>
<td>Display a list of all spied goals</td>
</tr>
<tr>
<td><code>defdetrel (+name &amp;rest +clauses)</code></td>
<td>Define a relation and declare it to be deterministic</td>
</tr>
<tr>
<td><code>defgrammar (+name &amp;rest +rules)</code></td>
<td>Define a grammar rule</td>
</tr>
<tr>
<td><code>defrel (+name &amp;rest +clauses)</code></td>
<td>Define a relation</td>
</tr>
<tr>
<td><code>defrelmacro (+name +args &amp;rest +body)</code></td>
<td>Define a logic macro</td>
</tr>
<tr>
<td><code>defrel-special-form-macro (+name +args &amp;rest +body)</code></td>
<td>Like <code>defrelmacro</code> but can have &amp;rest in <code>+args</code>. Use of this form will shadow all predicates named <code>+name</code> regardless of arity.</td>
</tr>
<tr>
<td><code>deterministic (+name)</code></td>
<td>Declare the relation called <code>?name</code> to be deterministic</td>
</tr>
<tr>
<td><code>erase (+ref)</code></td>
<td>Delete the predicate with database reference <code>?ref</code> from the database</td>
</tr>
<tr>
<td><code>fail()</code></td>
<td>Standard Prolog</td>
</tr>
<tr>
<td><code>findall (+exp +goal ?result)</code></td>
<td>Generate all solutions to <code>?goal</code> and instantiate <code>?exp</code> with the values. Return a list in <code>?result</code>.</td>
</tr>
<tr>
<td><code>findallset (+exp +goal ?result)</code></td>
<td>Same as <code>findall/3</code> but removes duplicates</td>
</tr>
<tr>
<td><code>halt()</code></td>
<td>Exit Common Prolog</td>
</tr>
<tr>
<td><code>integer (+x)</code></td>
<td>Standard Prolog</td>
</tr>
<tr>
<td><code>is (+result +exp)</code></td>
<td>Standard Prolog</td>
</tr>
<tr>
<td><code>keysort (+in ?out)</code></td>
<td>Standard Prolog except uses alist style cons pairs</td>
</tr>
<tr>
<td><code>leash (+event-spec)</code></td>
<td>Cause the interpreter to pause and ask for input when one of the leashed events is traced. An event-spec is one of: <code>call</code>, <code>exit</code>, <code>redo</code> or <code>fail</code>, or a list of ports.</td>
</tr>
<tr>
<td><code>listing (+name &amp;optional +arity)</code></td>
<td>Display a listing of the named predicate or listings for each arity if no arity is specified</td>
</tr>
<tr>
<td><code>member (+x ?y)</code></td>
<td>Standard Prolog</td>
</tr>
<tr>
<td><code>nodebug()</code></td>
<td>Leave debug mode (cease saving debug info for non-spied goals)</td>
</tr>
<tr>
<td><code>nonvar (+x)</code></td>
<td>Standard Prolog</td>
</tr>
<tr>
<td><code>nospy (+args)</code></td>
<td>Remove <code>+args</code> from the list of spied goals. <code>+args</code> may be a predicate name or a list of predicate names. Unspy all goals if <code>+args</code> is <code>nil</code></td>
</tr>
<tr>
<td><code>not (+x)</code></td>
<td>Standard Prolog</td>
</tr>
<tr>
<td><code>notrace()</code></td>
<td>Turn off exhaustive tracing for debugged goals</td>
</tr>
<tr>
<td>Procedure</td>
<td>Description</td>
</tr>
<tr>
<td>-----------------------------</td>
<td>-------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td><code>once (+exp)</code></td>
<td>satisfy <code>+exp</code> as a goal once, then fail on retrying even if <code>+exp</code> has more solutions: this can be used to make a call deterministic so that the compiler can perform last call optimization</td>
</tr>
<tr>
<td><code>output-defrels (+name ?defrels)</code></td>
<td>return a list of <code>defrel</code> expressions derived from the dynamic clauses associated with <code>?name</code></td>
</tr>
<tr>
<td><code>read-term (?term)</code></td>
<td>read in a term</td>
</tr>
<tr>
<td><code>recorda (+exp ?val ?ref)</code></td>
<td>standard Prolog</td>
</tr>
<tr>
<td><code>recordd (+term ?val ?ref)</code></td>
<td>standard Prolog</td>
</tr>
<tr>
<td><code>recordz (+exp ?val ?ref)</code></td>
<td>standard Prolog</td>
</tr>
<tr>
<td><code>repeat ()</code></td>
<td>standard Prolog</td>
</tr>
<tr>
<td><code>retract (+clause)</code></td>
<td>standard Prolog</td>
</tr>
<tr>
<td><code>setof (?exp (+goal . +ex-vars) ?bag)</code></td>
<td>standard Prolog (unusual syntax)*</td>
</tr>
<tr>
<td><code>sort (+in ?out)</code></td>
<td>standard Prolog</td>
</tr>
<tr>
<td><code>spy (+args)</code></td>
<td>spy <code>+args</code>. <code>+args</code> may be a predicate name or a list of predicate names. If arity is not mentioned for a predicate name, predicates of all arities with that name are spied.</td>
</tr>
<tr>
<td><code>trace ()</code></td>
<td>turn on tracing for debugged goals, also turn on debugging for the next top level goal</td>
</tr>
<tr>
<td><code>translate-vars (?intern ?extern)</code></td>
<td>translate back and forth between internal and external variable representations. Can be used to pretty up the writing of terms containing variables</td>
</tr>
<tr>
<td><code>true ()</code></td>
<td>standard Prolog</td>
</tr>
<tr>
<td><code>unleash (+event-spec)</code></td>
<td>Undo leashing for <code>+event-spec</code>. <code>+event-spec</code> may be a port or a list of ports. If <code>+event-spec</code> is <code>nil</code>, all ports are unleashed.</td>
</tr>
<tr>
<td><code>var (?x)</code></td>
<td>standard Prolog</td>
</tr>
<tr>
<td><code>phrase +exp ?list</code></td>
<td>standard Prolog way to call a grammar rule</td>
</tr>
<tr>
<td><code>phrase +exp ?list ?tail</code></td>
<td>standard Prolog way to call a grammar rule</td>
</tr>
</tbody>
</table>

* `setof` and `bagof` in standard Prolog use a special syntax for existentially quantified variables, for example:

\[-\text{setof}(X, \ Y^\text{foo}(X,Y), \ Z).\]

In Common Prolog, this would look like:

\[\text{==> (setof \ ?x \ ((\text{foo ?x ?y}) \ ?y) \ ?z)}\]

So, a goal with no existentially quantified variables is nested in an extra set of parentheses:

\[\text{==> (bagof \ ?x \ ((\text{bar ?x})) \ ?z)}\]
A.13 Adding Built-in Predicates

Common Prolog provides several special forms for adding new predicates written in Lisp. Each one is described below, with an example.

A.13.1 The defdetpred form

The syntax of this form is:

```
(defdetpred <name> <num-args> <body>)
```

which defines a simple predicate that just runs Lisp code and does not have to unify any variables. Arguments are referenced with: `(special-arg <num>)`. The body succeeds by default, but if a failure case arises, use: `(detpred-fail <name> <num-args>)`.

For example:

```
(defdetpred my-integer 1
  (unless (integerp (special-arg 0))
    (detpred-fail my-integer 1)))
```

A.13.2 The defdetunipred form

The syntax of this form is:

```
(defdetunipred <name> <num-args> <unifier1 unifier2>
  <aux-vars> <body>)
```

`defdetunipred` is used when the defined predicate needs to unify values with arguments (or unify in general). The body is executed and, if successful, (that is, `detpred-fail` has not been called) unification is performed on the two unifiers. (If more than two items need to be unified, cons up lists of items to unify).

For example:

```
(defdetunipred my-arg 3 (temp1 temp2)
  (temp1 temp2 index term value)
  (setf index (special-arg 0)
    term (special-arg 1)
    value (special-arg 2))
  (unless (and (numberp index)
    (plusp index)
    (or (and (term-p term)
      (< index (length term)))
    (and (consp term)
      (< index 3))))
    (detpred-fail my-arg 3))
  (if (consp term)
    (setf temp1 (if (= index 1)
      (car term)
      (cdr term)))
    (setf temp1 (term-ref term index)))
  (setf temp2 value))
```
A.14 Edinburgh Compatibility Predicates

The following predicates all have their standard Edinburgh definitions (note that these are written as Lisp symbols, so \ is a Lisp escape character, meaning that "\+") is the Edinburgh definition named "+":

```
-->
->
/ //
<< == == <<
>> ?- @< @>
>@= \, \.
\:= \; \`
|\+ \\| \\=
|\==\
^ current-op
display
get
get0
is
name
n1
put
see
seeing
seen
skip
tell
telling
told
ttynl
ttyput
write
writeq
|is|
```
Appendix B: Examples

B.1 The Tutorial

The code for the tutorial (2 Tutorial) is reproduced for easy reference.

; -*-mode : lisp ; package : kw-user -*-

(in-package kw-user)

;;; ---------------- OBJECT DEFINITIONS ------------
(def-kb-class node ()
  ((animal :initform nil :accessor node-animal
    :initarg :animal)
   (question :initform nil :accessor node-question
    :initarg :question)
   (yes-node :initform nil :accessor node-yes-node
    :initarg :yes-node)
   (no-node :initform nil :accessor node-no-node
    :initarg :no-node)))

(def-kb-class root ()
  ((node :initform nil :accessor root-node
    :initarg :node)))

(def-kb-struct current-node node)
(def-kb-struct game-over node animal answer)

;;; -------------- FORWARD CHAINING RULES -------------

;;; if there is no question we are about to ask then
;;; ask the question which is the root question of the
;;; question tree

(defrule play :forward
  (root ?r node ?node)
  (not (current-node ? node ?))
  -->
  ((tk:send-a-message
    (format nil " ANIMAL GUESSING GAME - -
              think of an animal to continue")
    (assert (current-node ? node ?node)))
  ;; ask a yes/no question - these are non-leaf questions

(defrule y-n-question :forward
  (current-node ?current node ?node)
  (node ?node animal nil question ?q yes-node ?y-n
     no-node ?n-n)
  -->
  ((tk:confirm-yes-or-no ?q) ?answer)
  (erase ?current)
  ((find-new-node ?answer ?y-n ?n-n) ?new-current)
  (assert (current-node ? node ?new-current)))

(defun find-new-node (answer yes-node no-node)
  (if answer yes-node no-node))
(defrule new-question :forward
  (game-over ? node ?node animal ?animal answer nil)
  -->
  (fetch-new-animal ?new-animal)
  ((tk:popup-prompt-for-string
      (format nil "Tell me a question for which the ~a answer is yes for a ~a and no for a ~a"
  (assert (node ?yes-node question nil animal ?new-animal))
  (assert (node ?no-node question nil animal ?animal))
  (assert (node ?node animal nil yes-node ?yes-node no-node ?no-node question ?question)))

;;; game is over
(defrule game-finished :forward
  (game-over ?g)
  -->
  (erase ?g)
  (test (not (tk:confirm-yes-or-no "Play again?")))
  (return))

;;; --------------- BACKWARD CHAINING ----------------

;;; prompt user for new animal
(defrule fetch-new-animal :backward
  ((fetch-new-animal ?new-animal)
   <--
   (repeat)
   (string-upcase
    (tk:popup-prompt-for-string
     "What was your animal?"
     ?new-animal)
    (not (= ?new-animal "NIL"))
    ; check if abort was pressed
    (or
     (does-not-exist-already ?new-animal)
     (and ((tk:send-a-message "Animal exists already")
           (fail)))))
  ;; check if a node already refers to this animal
(defrule does-not-exist-already :backward
  ((does-not-exist-already ?animal)
   <--
   (node ? animal ?animal)
   (cut)
(fail)
((does-not-exist-already ?animal)
  \=)

;;; ------------ SAVING THE ANIMAL BASE ------------

;;; writes out code which when loaded reconstructs the
;;; tree of questions

(defun save-animals (filename)
  (let* ((start-node (any `?node `(root ? node ?node)))
         (code `(make-instance `root
                :node ,(node-code start-node)))
         (*print-pretty* t))
    (with-open-file
      (stream filename :direction :output
        :if-exists :supersede)
      (write `(in-package kw-user) :stream stream)
      (write-char #\Newline stream)
      (write code :stream stream))
    nil))

(defun node-code (node)
  (when node
    `(make-instance `node
       :question ,(node-question node)
       :animal `,(node-animal node)
       :yes-node ,(node-code (node-yes-node node))
       :no-node ,(node-code (node-no-node node)))))

B.2 Explanation Facility

Below is the complete code implementing the simple explanation facility of 6.1.1.3 A Simple Explanation Facility. The implementation principle is exactly as described.

;;; ------------ A SIMPLE EXPLANATION FACILITY ------------

(in-package kw-user)

; connects rule to explanation definitions
(defvar *explanation-table*
  (make-hash-table :test #'eq))

; explanation generated at run time
(defvar *explanation* nil)

;;; the next four definitions make up the defexplain
;;; macro for each of the why, what and because
;;; definitions we create a function which we can call
;;; at run time on the bindings of the instantiation to
;;; generate the explanation text - this will be
;;; reasonably efficient

(defun is-var (expr)
  "is this a variable (i.e. starts with ?)"
  (and (symbolp expr)
       (eql (char (symbol-name expr) 0) #\?))

(defun find-vars (expr)
  "returns a list of all the variables in expr"
  (if (consp expr)
      (append (find-vars (car expr))
              (find-vars (cdr expr)))
      nil))
(if (is-var expr) (list expr) nil))

(defun make-explain-func (explain-stuff)
  "generates a function to generate explanation text at run time"
  (let* ((explain-string (car explain-stuff))
         (explain-args (cdr explain-stuff))
         (vars (remove-duplicates (find-vars explain-args)))
         `#'(lambda (bindings)
              (let ,(mapcar #'(lambda (v)
                                `(,v (cdr (assoc ,v bindings))))
                              vars)
                 (format nil ,explain-string ,@explain-args)))))

(defmacro defexplain (rulename &key why what because)
  "puts an entry for the rule in the explanation table"
  `(setf (gethash ',rulename *explanation-table*)
         (list ,(make-explain-func why)
               ,(make-explain-func what)
               ,(make-explain-func because))))

;;;; next two definitions generate an explanation for
;;;; each instantiation that fires and stores it away in
;;;; *explanation*

(defun add-explanation (inst)
  "generate an explanation for firing this instantiation"
  (let ((explain-info
          (gethash (inst-rulename inst)
                *explanation-table*))
        (when explain-info
         (do-the-rest explain-info (inst-bindings inst)))))

(defun do-the-rest (explain-info bindings)
  "creates explanation text derived from explain functions and bindings"
  (let ((why-func (first explain-info))
         (what-func (second explain-info))
         (because-func (third explain-info)))
    (push `(*cycle* ,(inst-rulename inst)
               ,(funcall why-func bindings)
               ,(funcall what-func bindings)
               ,(funcall because-func bindings)
               *explanation*)))))

;;;; meta-interpreter for explanation contexts
;;;; before firing the rule generate explanation for
;;;; this cycle

(defrule explain-context :backward
  (explain-context)
  (start-cycle)
  (instantiation ?inst)
  ((add-explanation ?inst))
  (fire-rule ?inst)
  (cut)
  (explain-context)))

;;;; simple text output of the explanation

(defun explain (&optional cycle)
"print out either the whole explanation or just for one cycle"
(if cycle (explain-cycle (assoc cycle *explanation*))
   (dolist (cycle-entry (reverse *explanation*))
     (explain-cycle cycle-entry))))

(defun explain-cycle (entry)
  "print this explanation entry"
  (if entry
      (let ((cycle (first entry))
          (rulename (second entry))
          (why (third entry))
          (what (fourth entry))
          (because (fifth entry)))
        (format t "~2%~a: ~a~%~a~%~a~%~a" cycle rulename why what because)
        (format t "~2%No explanation for this cycle")))

;;; we could make a really smart tool here, but to give
;;; the general idea...

(defun explain-an-action ()
  (let ((item
         (tk:scrollable-menu
          (reverse *explanation*)
          :title "Which action do you want explained?"
          :name-function #'(lambda (x) (fourth x))))))
    (if item (tk:send-a-message (fifth item))))

;;; starting the rule interpreter should clear any old
;;; explanation

(defadvice (infer rest-explanation :before)
  (&rest args)
  (unless *in-interpreter* (setq *explanation* nil)))

Below are some example rules using the explanation facility. They are taken from the Monkey and Banana Example distributed with KnowledgeWorks. The classes used in the example are monkey, object and goal.

(defrule mb7 :forward
  :context mab
  (goal ?g status active type holds object ?w)
  (object ?o1 kb-name ?w at ?p on floor)
  (monkey ?m at ?p holds nil)
  -->
  ((format t "-%Grab ~s" ?w))
  (assert (monkey ?m holds ?w))
  (assert (goal ?g status satisfied)))

(defexplain mb7
  :why ("Monkey is at the ~s which is on the floor" ?w)
  :what ("Monkey grabs the ~s" ?w)
  :because ("Monkey needs the ~s somewhere else" ?w))

(defrule mb12 :forward
  :context mab
  (goal ?g status active type walk-to object ?p)
  (monkey ?m on floor at ?c holds nil)
  (test (not (eq ?c ?p)))
  -->
  ((format t "-%Walk to ~s" ?p))
  (assert (monkey ?m at ?p))
  (assert (goal ?g status satisfied)))
(defexplain mb12
  :why ("Monkey is on the floor holding nothing")
  :what ("Monkey walks to ~s" ?p)
  :because ("Monkey needs to do something with an
            object at ~s" ?p))

(defrule mb13 :forward
  :context mab
  (goal ?g status active type walk-to object ?p)
  (monkey ?m on floor at ?c holds ?w)
  (test (and ?w (not (eq ?c ?p))))
  (object ?ol kb-name ?w)
  -->
  ((format t "Walk to ~s") ?p)
  (assert (monkey ?m at ?p))
  (assert (object ?ol at ?p))
  (assert (goal ?g status satisfied)))

(defexplain mb13
  :why ("Monkey is on the floor and is holding the ~s" ?w)
  :what ("Monkey walks to ~s with the ~s" ?p ?w)
  :because ("Monkey wants the ~s to be at ~s" ?w ?p))

(defrule mb14 :forward
  :context mab
  (goal ?g status active type on object floor)
  (monkey ?m on ?x)
  (test (not (eq ?x `floor)))
  -->
  ((format t "Jump onto the floor")
   (assert (monkey ?m on floor))
   (assert (goal ?g status satisfied)))

(defexplain mb14
  :why ("Monkey is on ~s" ?x)
  :what ("Monkey jumps onto the floor")
  :because ("Monkey needs to go somewhere")

(defrule mb17 :forward
  :context mab
  (goal ?g status active type holds object nil)
  (monkey ?m holds ?x)
  (test ?x)
  -->
  ((format t "Drop ~s") ?x)
  (assert (monkey ?m holds nil))
  (assert (goal ?g status satisfied)))

(defexplain mb17
  :why ("Monkey is at the location of the ~s" ?o)
  :what ("Monkey climbs onto the ~s") ?o)
  :because ("Monkey wants to be on top of the ~s" ?o))

(defrule mb18 :forward
  :context mab
  (goal ?g status active type holds object nil)
  (monkey ?m holds ?x)
  (test ?x)
  -->
  ((format t "Drop ~s") ?x)
  (assert (monkey ?m holds nil))
  (assert (goal ?g status satisfied)))

(defexplain mb18
Appendix B: Examples

:why ("Monkey is holding the ~s" ?x)
:what ("Monkey drops the ~s" ?x)
:because ("Monkey wants to do something for which he can't hold anything")

B.3 Uncertain Reasoning Facility

Below is the complete code which implements the uncertain reasoning facility of 6.1.1.4 Reasoning with Certainty Factors. The implementation is exactly as described with a few extra considerations to check the rule interpreter is running before returning an uncertain value, that the objects have a certainty-factor slot and so on.

;;; -----SIMPLE REASONING WITH UNCERTAINTY FACTORS ----
(in-package kw-user)

;;; default certainty factor
(defvar *c-factor* 1)

;;; implication strength of a rule
(defvar *implication-strength* 1)

(defun default-c-factor ()
  "if the forward chainer is not running, certainty factor is just 1"
  (if *in-interpreter*
      (* *implication-strength* *c-factor*)
      1))

;;; uncertain objects need a slot to store their
;;; 'probability' this slot defaults to the value
;;; returned by default-c-factor

(def-kb-class uncertain-kb-object ()
  ((c-factor :initform (default-c-factor)
             :initarg :c-factor)))

(defun object-c-factor (obj)
  "if an object has no uncertainty slot, return 1 (i.e. certain)"
  (if (slot-exists-p obj `c-factor)
      (slot-value obj `c-factor)
      1))

(defun inst-c-factor (inst)
  "the certainty factor of an instantiation"
  (token-c-factor (inst-token inst)))

(defun token-c-factor (token)
  "the certainty factor of an ANDed list of objects
  (just multiply them)"
  (reduce "*" (mapcar `object-c-factor token)))

(defun implication-strength (val)
  "for a rule to set the implication strength"
  (setq *implication-strength* val))

;;; this function increases the certainty of the object
;;; which is the first argument by an amount dependent
;;; on the combined certainty of the remaining
;;; arguments

(defun add-evidence (obj &rest token)
  "increments the certainty of obj based on the"
certainty of token
(let ((c-f (slot-value obj 'c-factor)))
  (setf (slot-value obj 'c-factor)
        (+ c-f
           (* (- 1 c-f) *implication-strength*
               (token-c-factor token))))))

;;; this tactic is dynamic as the certainty factor slot
;;; gets changed by calling add-evidence
(deftactic certainty :dynamic (i1 i2)  
  "a conflict resolution tactic to prefer more certain
  instantiations"
  (> (inst-c-factor i1) (inst-c-factor i2)))

;;; Before firing a rule this meta-interpreter just
;;; sets the value of *c-factor* to the certainty of
;;; the instantiation so that any new uncertain objects
;;; made get this (times *implication-strength*) as
;;; their certainty. Also sets *implication-strength*  
;;; to 1 as a default in case the rule does not set it.
(defrule uncertain-context :backward
  ((uncertain-context)
    (start-cycle)
    (instantiation ?inst)
    ((progn (setq *c-factor* (inst-c-factor ?inst))
             (setq *implication-strength* 1))
      (fire-rule ?inst)
      (cut)
      (uncertain-context))))

Below are some example rules using this facility for a simple car maintenance problem.

;;; ---------------- SOME EXAMPLE RULES ---------------
;;; to run: (run-diagnose)
(def-kb-struct start)
(def-kb-class symptom (uncertain-kb-object)
  ((type :initarg :type)))
(def-kb-class fault (uncertain-kb-object)
  ((type :initarg :type)))
(def-kb-class remedy (uncertain-kb-object)
  ((type :initarg :type)))

;;; this context sets up the initial hypotheses and
;;; gathers evidence this does not need the meta
;;; -interpreter as that’s only necessary for
;;; transparent assignment of certainty factors to new
;;; objects
(defcontext diagnose :strategy ()
(defrule start-rule :forward
  :context diagnose
  (start ?s)
  -->
  (assert (symptom ? type over-heat c-factor 1))
  (assert (symptom ? type power-loss c-factor 1))
  (assert (fault ? type lack-of-oil c-factor 0.5))
  (assert (fault ? type lack-of-water c-factor 0))
  (assert (fault ? type battery c-factor 0))
  (assert (fault ? type unknown c-factor 0))
  (context (cure)))
(defrule diagnose1 :forward
:context diagnose
(sympotm ?s type over-heat)
(fault ?f type lack-of-water)
--> 
((implication-strength 0.9))
((add-evidence ?f ?s)))

(defrule diagnose2 :forward
:context diagnose
(sympotm ?s type overheat)
(fault ?f type unknown)
--> 
((implication-strength 0.1))
((add-evidence ?f ?s)))

(defrule diagnose3 :forward
:context diagnose
(sympotm ?s type wont-start)
(fault ?f type battery)
--> 
((implication-strength 0.9))
((add-evidence ?f ?s)))

(defrule diagnose4 :forward
:context diagnose
(sympotm ?s type wont-start)
(fault ?f type unknown)
--> 
((implication-strength 0.1))
((add-evidence ?f ?s)))

(defrule diagnose5 :forward
:context diagnose
(sympotm ?s type power-loss)
(fault ?f type lack-of-oil)
--> 
((implication-strength 0.9))
((add-evidence ?f ?s)))

(defrule diagnose6 :forward
:context diagnose
(sympotm ?s type power-loss)
(fault ?f type unknown)
--> 
((implication-strength 0.1))
((add-evidence ?f ?s)))

;;; any two distinct symptoms strengthens the hypothesis that there's something more serious going wrong

(defrule diagnose7 :forward
:context diagnose
(sympotm ?s1 type ?t1)
(sympotm ?s2 type ?t2)
(test (not (eq ?t1 ?t2)))
(fault ?f type unknown)
--> 
((add-evidence ?f ?s1 ?s2)))

;;; here we need the meta-interpreter to assign the right certainty factors to the remedy objects. Also use certainty as a conflict resolution tactic to
(defcontext cure :strategy (priority certainty) :meta ((uncertain-context)))

(defrule cure1 :forward :context cure (fault ?f type unknown) ---> ((implication-strength 0.1)) (assert (remedy ? type cross-fingers)) ((implication-strength 0.9)) (assert (remedy ? type go-to-garage)))

(defrule cure2 :forward :context cure (fault ?f type lack-of-oil) ---> (assert (remedy ? type add-oil)))

(defrule cure3 :forward :context cure (fault ?f type lack-of-water) ---> (assert (remedy ? type add-water)))

(defrule cure4 :forward :context cure (fault ?f type battery) ---> (assert (remedy ? type new-battery)))

(defrule print-cures :forward :context cure :priority 5 (remedy ?r type ?t) ---> ((format t "Suggest remedy ~a with certainty-factor ~a" ?t (slot-value ?r `c-factor))))

(defun run-diagnose ()
(reset)
(make-instance `start)
(infer :contexts `(diagnose)))

B.4 Other Examples

Other examples distributed with KnowledgeWorks include:

- Truck — a largely forward chaining truck scheduling example.
- Spill — an outline of a chemical spillage diagnosis system.
- Whist — a windowing example which plays whist.
Appendix C: Implementation Notes

C.1 Forward Chainer

C.1.1 Forward Chaining Algorithm

The KnowledgeWorks forward chaining engine is based on the RETE algorithm (see Rete: A Fast Algorithm for the Many Pattern/Many Object Pattern Match Problem by Forgy in Artificial Intelligence 19, September 1982). A data flow network representing the conditions of the forward chaining rules (a RETE network) is maintained and this keeps lists of the instantiations and partial instantiations of rules. This structure is modified at run time as objects change. The RETE algorithm relies on the tacit assumption that during the forward chaining cycle relatively few objects change (hence there are relatively few changes to be made to the network each cycle), and in these cases gives a huge increase in performance speed.

C.1.2 CLOS and the Forward Chainer

CLOS objects acquire KnowledgeWorks functionality from the standard-kb-object mixin. Object creation and modification hooks defined on this mixin enable the RETE network to track the objects. Objects are indexed into the RETE network by class and modifications propagated only where any changes to the slots of the object are relevant.

One potential problem is that as KnowledgeWorks CLOS objects are designed for use in ordinary code, performance could deteriorate seriously as every time an object is changed the RETE network must be amended. For this reason changes to CLOS objects are merely remembered as they are made. The stored set of changes is flushed at the start of every forward chaining cycle, so the penalty for using KnowledgeWorks objects is really only paid when the forward chainer is running.

C.1.3 Forward Chaining and the Backward Chainer

For more uniform semantics throughout KnowledgeWorks, the right hand side of KnowledgeWorks forward chaining rules are executed directly by the backward chainer, as is the default meta-interpreter for a context which has no meta-interpreter specially defined. When compiled with debugging turned off, in many cases the backward chainer can be optimized out leaving raw Lisp code.

C.2 Backward Chainer

C.2.1 Backward Chaining Algorithm

The KnowledgeWorks backward chaining system is an extended Prolog written entirely in Lisp and based loosely on the Warren Abstract Machine (WAM). (see An Abstract Prolog Instruction Set by David H.D. Warren, Technical Note 309 SRI International October 1983). High performance is achieved by compiling each Prolog clause into a Lisp function and handling the Prolog control flow with continuation passing. This approach removes the need for interpretation and provides easy integration with CLOS.
C.2.2 Term Structure

In order to provide compatibility with Edinburgh Prolog, the KnowledgeWorks backward chaining system treats Prolog structured terms differently from lists. Structured terms whose functors are not `.` are stored as simple vectors with the functor as element 0 (for example, the term: \texttt{foo(bar)} is equivalent to \#(foo bar)).

C.2.3 The Binding Trail

The variable binding trail for the backward chainer is stored in a simple vector but may overflow into list structure if the trail grows larger than the size of the vector: (30000). The system will continue to function normally when this happens but may slow down slightly and do more consing. (Note: We have never written a program that causes this to happen other than deliberately produced testing programs).
Appendix D: For More Information

D.1 General References

D.1.1 Forward Chaining

• *Programming Expert Systems in OPS5, An Introduction to Rule-Based Programming* by Lee Brownston, Robert Farrell, Elaine Kant and Nancy Martin (Addison-Wesley). While being specifically on OPS5, this text covers most aspects of forward chaining in considerable detail.

D.1.2 Backward Chaining and Prolog

• *The Art of Prolog*, by Leon Sterling and Ehud Shapiro (MIT Press).

• *The Craft of Prolog*, by Richard A. O'Keefe (MIT Press). This is a more advanced text.

D.1.3 Uncertain Reasoning

• *Rule-Based Expert Systems*, by B. G. Buchanan and E. H. Shortliffe (Addison-Wesley). This text covers specifically the MYCIN system.

D.1.4 Expert Systems

• *Building Expert Systems*, by Frederick Hayes-Roth, Donald A. Waterman and Douglas B. Lenat (Addison-Wesley). This text focuses more on the issues involved in designing an expert system.

D.1.5 Lisp and CLOS


• *The Art of the Metaobject Protocol*, by Gregor Kiczales, Jim des Rivieres and Daniel G. Bobrow (MIT Press). This is the only proper guide to the CLOS Metaobject Protocol.

D.2 The LispWorks manuals

In addition to the KnowledgeWorks and Prolog User Guide, the LispWorks manual set includes the following manuals which might be helpful while using KnowledgeWorks:

• The *LispWorks® User Guide and Reference Manual* describes the language-level features and tools available in LispWorks, along with detailed information on the functions, macros, variables and classes.
Appendix D: For More Information

- The *LispWorks IDE User Guide* describes the LispWorks IDE, the user interface for LispWorks. the LispWorks IDE is a set of windowing tools that let you develop and test Common Lisp code more easily and quickly.

- The *Editor User Guide* describes the keyboard commands and programming interface to the the LispWorks IDE editor tool.

- The *Release Notes and Installation Guide* explains how to install LispWorks, configure it and start it running. It also contains a set of release notes that documents last minute issues that could not be included in the main manual set.

These books are all available in HTML and PDF formats.

Commands in the *Help* menu of any of the the LispWorks IDE tools give you direct access to the online documentation in HTML format. Details of how to use these commands can be found in the *LispWorks IDE User Guide*.

Please let us know at *lisp-support@lispworks.com* if you find any mistakes in the LispWorks documentation, or if you have any suggestions for improvements.
Appendix E: Converting Other Systems

E.1 OPS5

OPS5 rulebases may be readily converted into KnowledgeWorks rulebases. The main OPS5 forms needing conversion are:

- **literalize** into `def-kb-struct` or `def-kb-class`. For example:

  (literalize employee name father-name mother-name)

  could become:

  (def-kb-struct employee name father-name mother-name)

- **strategy** into a defcontext form with the right conflict resolution strategy. For example:

  (strategy lex)

  could become:

  (defcontext ops5 :strategy (lex specificity))

  and:

  (strategy mea)

  could become:

  (defcontext ops5 :strategy (mea lex specificity))

In OPS5 you cannot have different conflict resolution strategies for different sets of rules. The KnowledgeWorks context mechanism for passing control is much clearer and more powerful than, for instance, the use of the MEA strategy as sole control mechanism in OPS5.

- **p** into `defrule`. For example, the OPS5 rule:

  (p recognize-pair
   (employee ^name <parent>)
   (employee ^name <child> ^mother-name <parent>)
   --->
   (make pair))

  will become:

  (defrule recognize-pair :forward
   (employee ? name ?parent)
   (employee ? name ?child mother-name ?parent)
   --->
   (assert (pair ?)))

As an extended example below are given some OPS5 rules from the Monkey and Banana problem (see Appendix B:  

144
Examples):

```
(strategy mea)
(literalize monkey
  name at on holds)
(literalize object
  name at weight on)
(literalize goal
  status type object to)
(literalize start)

(p mb1
  (goal ^status active ^type holds ^object <w>)
  (object ^name <w> ^at <p> ^on ceiling)
  --->
  (make goal ^status active ^type move ^object ladder
    ^to <p>))

(p mb4
  {(goal ^status active ^type holds ^object <w>) <goal>}
  (object ^name <w> ^at <p> ^on ceiling)
  (object ^name ladder ^at <p>)
  {(monkey ^on ladder ^holds nil) <monkey>}
  --->
  (write (crlf) Grab <w>)
  (modify <goal> ^status satisfied)
  (modify <monkey> ^holds <w>))

(p mb8
  (goal ^status active ^type move ^object <o> ^to <p>)
  (object ^name <o> ^weight light ^at <> <p>)
  --->
  (make goal ^status active ^type holds ^object <o>))
```

In KnowledgeWorks this could be:

```
(defcontext ops5 :strategy (mea lex specificity))

(def-named-kb-class monkey ()
  ((at :initform nil)
   (on :initform nil)
   (holds :initform nil)))

(def-named-kb-class object ()
  ((at :initform nil)
   (weight :initform nil)
   (on :initform nil)))

(def-kb-struct goal status type object to)
(def-kb-struct start)

(defrule mb1 :forward
  :context ops5
  (goal ? status active type holds object ?w)
  (object ? name ?w at ?p on ceiling)
  --->
  (assert (goal ? status active type move object ladder
to ?p)))

(defrule mb4 :forward
  :context ops5
  (goal ?g status active type holds object ?w)
  (object ? name ?w at ?p on ceiling)
  (object ? name ladder at ?p)
  (monkey ?m on ladder holds nil)
Appendix E: Converting Other Systems

---

((format t "~%Grab ~S" ?w))
(assert (goal ?g status satisfied))
(assert (monkey ?m holds ?w)))

(defrule mb8 :forward
 :context ops5
 (goal ? status active type move object ?o to ?p)
 (object ? name ?o weight light at ?q)
 (test (not (eq ?q ?p)))
 -->
 (assert (goal ? status active type holds object ?o)))

E.2 Prolog

Please refer to A.10 Edinburgh Syntax.
Glossary

agenda
A stack of rule groups (or contexts). Control can be passed to the next context on the agenda.

arity
The number of arguments (to a function, rule condition etc.)

backward chaining
The process of reasoning backward from postulated goals to determine if their preconditions can be satisfied. If these preconditions are satisfied the postulated goals are considered true.

browsers
Windows which allow you to look freely through different parts of the system.

class
In object-oriented programming, classes define classes with the same attributes (slots) and behavior (methods). Instances of these classes are created during the execution of a program which represent concrete examples of the abstract class descriptions.

conflict resolution strategy
The method(s) used to decide which of a set of eligible rules will fire. A conflict resolution strategy is a list of conflict resolution tactics which are applied in sequence to the conflict set to determine which instantiation is to fire.

conflict resolution tactic
A single predicate used to decide whether one instantiation is to be preferred to another. They may be combined into a conflict resolution strategy.

conflict set
The set of instantiations of rules which at a given time are matched by the object base.

contexts
Groups of rules in a knowledge base.

destructuring
The ability to match an expression against a piece of data where variables in the expression are bound to the corresponding parts of the data if the structure of the expression and the data agree. For example, (?x . ?y) can match (1 2 3) with ?x binding to 1 and ?y to (2 3).

forward chaining
The process of reasoning forward from known facts to perform arbitrary actions and to deduce new facts.
Glossary

**forward chaining cycle**

The process of matching the conditions of rules against the object base to produce a set of rules eligible to fire (the conflict set), selecting one of those (conflict resolution) and firing it (performing its actions).

**inference engine**

The part of the system which is responsible for rule-firing, either in backward or forward chaining mode.

**inferencing state**

A collection of information that the inferencing engine uses.

**instantiation**

An instantiation of a rule is the set of objects against which a rule matches. A rule may have no instantiations (if it is not matched at all by the object base) or many instantiations (each referring to a different set of objects).

**knowledge based systems**

A system which encodes the knowledge for a problem domain in high-level forms, usually facts and rules. The software architecture separates the knowledge from the inference mechanism used to deduce new knowledge.

**LispWorks**

An advanced Common Lisp programming environment, which serves as the infrastructure for KnowledgeWorks.

**meta object protocol (MOP)**

Describes how the Common Lisp Object System is implemented in terms of itself. Hence CLOS may be used to modify its own behavior.

**meta rule protocol (MRP)**

Allows you to debug, modify or replace the default behavior of forward chaining rules in the system in terms of backward chaining goals.

**object base**

The set of CLOS objects which KnowledgeWorks can reason over ("knows about").

**object-oriented**

Programming paradigm in which structures within the language are organized as classes of objects which have attributes (slots) and behavior (methods) associated with them.

**objects**

The KnowledgeWorks® object base contains KnowledgeWorks CLOS objects, which may for efficiency be replaced by KnowledgeWorks structures.

**structures**

A CLOS class can be replaced by a structure class in cases where speed is important and the code must be optimized, and when the full power of CLOS is not required. The structure is then analogous to the CLOS object.
toolkit

A collection of complementary software or utilities (such as KnowledgeWorks®) with a common application focus.
Index

A

action 1.1.2: Technical Overview 9, 3.1.1: Overview 23, 3.2.5: The Backward Chaining Interpreter 29

add-explanation 6.1.1.3: A Simple Explanation Facility 51

Advanced Topics

main chapter 6: Advanced Topics 49
agenda 3.1.5.1: The Agenda 25

all-debug function 61 3.1.7: Forward Chaining Debugging 27
any function 61 3.2.5: The Backward Chaining Interpreter 28, A.6.2.1: any, findall and findallset 116
append A.12: Built-in Predicates 125
arg A.2: Syntax 111, A.12: Built-in Predicates 125
arity 3.2.2: Backward Chaining Syntax 28, 5.1: The KnowledgeWorks Listener 37
assert 3.1.2: Forward Chaining Syntax 24
assert backward chaining goal 62 6.4: Logical Dependencies and Truth Maintenance 58
asserta A.12: Built-in Predicates 125
assertion 3.1.2: Forward Chaining Syntax 24
assertz A.12: Built-in Predicates 125
atomic A.12: Built-in Predicates 125

B

backward chaining 1.1.2: Technical Overview 8, 2.4.1: Rule Browser 14, 3: Rules 23, 3.2: Backward Chaining 27, 6.2.3: Backward Chaining 56
debugging 3.2.7: Backward Chaining Debugging 29
definition of rules 3.2.4: Defining Backward Chaining Rules 28
implementation notes  C.2: Backward Chainer 140
interpreter 3.2.5: The Backward Chaining Interpreter 28
syntax 3.2.2: Backward Chaining Syntax 28

backward chaining goals

assert 62 6.4: Logical Dependencies and Truth Maintenance 58

conflict-set 64
cut 66
erase 75 6.4: Logical Dependencies and Truth Maintenance 58
fail 75
fire-rule 78 6.1.1: Meta Rule Protocol 49
instantiation 82 6.1.1: Meta Rule Protocol 49, 6.1.1: Meta Rule Protocol 50
kw-class 86
not 94
Index

Common Lisp Object System (CLOS) 1.1.1: Background 8, 2: Tutorial 11
Common Prolog main chapter Appendix A: Common Prolog 110

condition 3.1.1: Overview 23
   syntax 3.1.2: Forward Chaining Syntax 23
conflict resolution 2.4.4: Forward Chaining History 17, 3.1.1: Overview 23, 3.1.5.2: Contexts 26, 3.1.5.3: Conflict Resolution 26
   optimizing 6.2.2: Conflict Resolution 56
   strategy 6.1.2: User-definable Conflict Resolution 53
   tactics 3.1.5.3: Conflict Resolution 26, 6.1.2: User-definable Conflict Resolution 53, 6.2.2.2: Optimization of the Strategy 56
   use of contexts 6.2.2.1: Use of Contexts 56
   user definable 6.1.2: User-definable Conflict Resolution 53
conflict resolution strategy 6.1.2: User-definable Conflict Resolution 53
conflict resolution tactic 3.1.5.3: Conflict Resolution 26, 6.1.2: User-definable Conflict Resolution 53
conflict resolution tactic / functions

  lex 87 3.1.5.3: Conflict Resolution 26
  -lex 88 3.1.5.3: Conflict Resolution 26, 6.2.2.2: Optimization of the Strategy 56
  mea 91 3.1.5.3: Conflict Resolution 26
  -mea 92 3.1.5.3: Conflict Resolution 26, 6.2.2.2: Optimization of the Strategy 56
  order 95 3.1.5.2: Contexts 26, 3.1.5.3: Conflict Resolution 26
  -order 96 3.1.5.3: Conflict Resolution 26
  priority 97 3.1.5.2: Contexts 26, 3.1.5.3: Conflict Resolution 26
  -priority 98 3.1.5.3: Conflict Resolution 26
  recency 99 3.1.5.2: Contexts 26, 3.1.5.3: Conflict Resolution 26
  -recency 100 3.1.5.3: Conflict Resolution 26, 6.2.2.2: Optimization of the Strategy 56
  specificity 102 3.1.5.3: Conflict Resolution 26
  -specificity 103 3.1.5.3: Conflict Resolution 26

conflict-set backward chaining goal 64
context 3.1.2: Forward Chaining Syntax 24, 6.1.1: Meta Rule Protocol 49
context backward chaining goal 65
context definition 3.1.6.1: Defining Contexts 27
contexts 2.4.1: Rule Browser 13, 3.1.5.1: The Agenda 25, 3.1.5.2: Contexts 26
control
   flow of 3.1.5: Control Flow 25, 6.1: Control Flow 49
creep 2.6.2: Single-Stepping Rules 19, 5.8.1: Spy Windows 45

current-cycle function 66
cut 6.2.3.3: Cut 57
cut backward chaining goal 66
cycle
   of forward chaining 3.1.1: Overview 23
*cycle* symbol macro 67
Index

D
DCG A.9 : Defining Definite Clause Grammars 122
def A.12 : Built-in Predicates 126
defbug in Prolog A.7 : Debugging 118
debugging 1.1.2 : Technical Overview 8, 2.6 : Debugging 18, 3.1.7 : Forward Chaining Debugging 27, 5.8 : Debugging with the Environment 45, A.12 : Built-in Predicates 126
backward chaining 3.2.7 : Backward Chaining Debugging 29
forward chaining 3.1.7 : Forward Chaining Debugging 27
default-context 3.1.4 : The Forward Chaining Interpreter 25, 5.3 : Clearing KnowledgeWorks 39, clear-all 64, clear-rules 64
defclass macro def-kb-class 69, def-named-kb-class 71
defclass macro in LispWorks 4.1.2 : Named Classes 30
defcontext macro 67, 3.1.5.2 : Contexts 26, 3.1.5.3 : Conflict Resolution 26, 3.1.6 : Examples 27, 6.1.1 : Meta Rule Protocol 49
defdetpred A.13.1 : The defdetpred form 128
defdetrel A.12 : Built-in Predicates 126
defdetunipred A.13.2 : The defdetunipred form 128
defexplain 6.1.1.3 : A Simple Explanation Facility 51, 6.1.1.3 : A Simple Explanation Facility 51
Defining Contexts 3.1.6.1 : Defining Contexts 27
Definite Clause Grammars A.9 : Defining Definite Clause Grammars 122
def-kb-class macro 69, 4.1.1 : Unnamed Classes 30, 4.1.2 : Named Classes 30, 6.1.1.4 : Reasoning with Certainty Factors 52, 6.3.1 : Example 57, 6.4.1 : Example 58
def-kb-struct macro 70, 6.2.1.1 : KnowledgeWorks Structures 55
def-named-kb-class macro 71, 4.1.2 : Named Classes 30
defrel A.2 : Syntax 110, A.12 : Built-in Predicates 126
defrelmacro A.8 : Common Prolog Macros 122, A.12 : Built-in Predicates 126
defrel-special-form-macro A.12 : Built-in Predicates 126
defrule macro 72, 3 : Rules 23, 3.1.2.1 : Example 25, 3.2.2.1 : Example 28, 6.1.1 : Meta Rule Protocol 49, 6.1.1.4 : Reasoning with Certainty Factors 52, 6.2.3.1 : Pattern Matching 56, 6.2.3.2 : Tail Recursion 57, 6.3.1 : Example 58, 6.4.1 : Example 58, 6.5.2 : The Current Inferencing State 59
defstruct macro 6.2.1.1 : KnowledgeWorks Structures 55
deftactic macro 73, 6.1.2 : User-definable Conflict Resolution 53
def-view-class in LispWorks 4.2.1 : Example 31
destroy-inferencing-state function 74, 6.5.1 : Creating and Maintaining Inferencing States 59
deterministic A.12 : Built-in Predicates 126
documentation strings 3 : Rules 23, 3.1.5.2 : Contexts 26, 6.1.2 : User-definable Conflict Resolution 53, defcontext 68, defrule 72, deftactic 73
dynamic conflict resolution 6.2.2.2 : Optimization of the Strategy 56

E
Edinburgh Prolog 3.2.6 : Edinburgh Prolog Translator 29
Index

infer 80  2.3: Running the Tutorial 12,  2.6.2: Single-Stepping Rules 19,  3.1.4: The Forward Chaining Interpreter 25,  6.5.2: The Current Inferencing State 59

inferencing-state-name 81

inst-bindings 83  6.1.1.1: Functions defined on Instantiations 50

inst-rulename 84  6.1.1.1: Functions defined on Instantiations 50

inst-token 84  6.1.1.1: Functions defined on Instantiations 50

list-all-inferencing-states 89  6.5.1: Creating and Maintaining Inferencing States 59

make-inferencing-state 89  6.5.1: Creating and Maintaining Inferencing States 59

no-debug 94  3.1.7: Forward Chaining Debugging 27

reset 101  5.3: Clearing KnowledgeWorks 39,  6.5.2: The Current Inferencing State 59

start-kw 106

functor A.2: Syntax 111,  A.12: Built-in Predicates 126

G

generic functions

kb-name 85  4.1.2: Named Classes 30

make-instance 90  4.1: CLOS objects 30,  4.1.1: Unnamed Classes 30,  4.1.2: Named Classes 30,  6.2.1.1: KnowledgeWorks Structures 55,  6.4: Logical Dependencies and Truth Maintenance 58

validate-superclass 6.3: Use of Meta-Classes 57

get-kb-object function 79  4.1.2: Named Classes 30

goals 1.1.2: Technical Overview 9

graphical tools 1.1.2: Technical Overview 8

H

halt A.12: Built-in Predicates 126

history 2.4.4: Forward Chaining History 17

  forward chaining 2.4.4: Forward Chaining History 17,  5.9.1: Forward Chaining History 47

I

Implementation Notes

appendix Appendix C: Implementation Notes 140

implication strength 6.1.1.4: Reasoning with Certainty Factors 52

*implic-strength* 6.1.1.4: Reasoning with Certainty Factors 52,  6.1.1.4: Reasoning with Certainty Factors 53

infer function 80  2.3: Running the Tutorial 12,  2.6.2: Single-Stepping Rules 19,  3.1.4: The Forward Chaining Interpreter 25,  6.5.2: The Current Inferencing State 59

inference engine 1.1.2: Technical Overview 9

*inferencing-state* variable 80  6.5.2: The Current Inferencing State 59

inferencing-state-name function 81

inferencing states

  creating and maintaining 6.5.1: Creating and Maintaining Inferencing States 59

  current 6.5.2: The Current Inferencing State 59,  *inferencing-state* 80

  definition of 6.5: Inferencing States 59

  interleaved 6.5.3.2: Interleaved in a Single Thread 60

  multiple threads 6.5.3.1: Multiple threads 59

155
Index

uses 6.5.3: Uses of Inferencing States 59

*in-interpreter* variable 82

inspector

instances 5.5: The Class Browser 40

instantiation backward chaining goal 82 6.1.1: Meta Rule Protocol 49, 6.1.1: Meta Rule Protocol 50

instantiations 6.1.1.1: Functions defined on Instantiations 50

inst-bindings function 83 6.1.1.1: Functions defined on Instantiations 50

inst-rulename function 84 6.1.1.1: Functions defined on Instantiations 50

inst-token function 84 6.1.1.1: Functions defined on Instantiations 50

integer A.12: Built-in Predicates 126

interface functions in Prolog A.6.2: Interface Functions 116

interpreter

backward chaining 3.2.5: The Backward Chaining Interpreter 28

forward chaining 3.1.4: The Forward Chaining Interpreter 25

Introduction

main chapter 1: Introduction 8

is A.12: Built-in Predicates 126

K

kb-name generic function 85 4.1.2: Named Classes 30

:kb-name initarg 4.1.2: Named Classes 30, named-kb-object 93

keysor A.12: Built-in Predicates 126

keyword

:backward 3: Rules 23

:forward 3: Rules 23

:meta 6.1.1: Meta Rule Protocol 49

:priority 2.4.4: Forward Chaining History 17

Knowledge Based Systems (KBS) 1.1.1: Background 8, 1.1.2: Technical Overview 9

KnowledgeWorks 1.1: KnowledgeWorks 8

backward chaining engine 3.2.1: Overview 27

clearing 5.3: Clearing KnowledgeWorks 39

CLOS objects 3.2.1: Overview 27, 4: Objects 30

Converting Other Systems Into, Appendix E: Converting Other Systems 144

generic functions 5.5: The Class Browser 42

historical perspective 1.1.1: Background 8

inspector 5.5: The Class Browser 40

instances 5.5: The Class Browser 40

listener 5.1: The KnowledgeWorks Listener 37

loading files 2.2: Loading the Tutorial 12

mixin class 4.2: Relational Database Objects 31

object base 3.2.1: Overview 27

objects 3.2.3: Objects 28

156
Index

podium 2.1: Getting Started 11
rule development 5: The Programming Environment 37
rule monitor 5.7: The Rule Browser 44, 5.9: Monitor Windows 45
rules in 3: Rules 23
running the tutorial 2.3: Running the Tutorial 12
spy window 5.7: The Rule Browser 44
structures 4.3: KnowledgeWorks Structures 36, 6.2.1.1: KnowledgeWorks Structures 54
technical overview 1.1.2: Technical Overview 8
tools 2.6.2: Single-Stepping Rules 19
Use of Meta-Classes 6.3: Use of Meta-Classes 57
kw-class backward chaining goal 86

L
leap 2.6.2: Single-Stepping Rules 19, 5.8.1: Spy Windows 45
leash A.12: Built-in Predicates 126
leashing A.7: Debugging 118
lex conflict resolution tactic / function 87 3.1.5.3: Conflict Resolution 26
 lex conflict resolution tactic / function 88 3.1.5.3: Conflict Resolution 26, 6.2.2.2: Optimization of the Strategy 56
Lisp 1.1.1: Background 8, 2: Tutorial 11, 3.1.2: Forward Chaining Syntax 24, 3.3: Common Lisp Interface 29, 4: Objects 30
integration of 2.7: Lisp Integration 20
LispWorks 1.1: KnowledgeWorks 8, 2: Tutorial 11, 2.8: Systems 21, 3.2.1: Overview 27
accessing Lisp from Prolog A.5: Accessing Lisp From Common Prolog 113
availability in KnowledgeWorks 5: The Programming Environment 37
calling Prolog A.6: Calling Prolog From Lisp 114
Common Prolog Logic Listener 5.1: The KnowledgeWorks Listener 37
SQL interface 4.2: Relational Database Objects 31
LispWorks IDE 2.7.1: The LispWorks IDE 21
list-all-inferencing-states function 89 6.5.1: Creating and Maintaining Inferencing States 59
listener 2.3: Running the Tutorial 12, 5.1: The KnowledgeWorks Listener 37
listing A.12: Built-in Predicates 126
loading files 2.2: Loading the Tutorial 12
logic A.6: Calling Prolog From Lisp 114


logical 3.1.2: Forward Chaining Syntax 24, 6.4: Logical Dependencies and Truth Maintenance 58
logic interpreter A.4: Using The Logic Interpreter 111
logic listener A.11: Graphic Development Environment 125

M
macro in Prolog A.8: Common Prolog Macros 122
macros
defclass def-kb-class 69, def-named-kb-class 71
defcontext 67 3.1.5.2: Contexts 26, 3.1.5.3: Conflict Resolution 26, 3.1.6: Examples 27, 6.1.1: Meta Rule Protocol 49
def-kb-class 69 4.1.1: Unnamed Classes 30, 4.1.2: Named Classes 30, 6.1.1.4: Reasoning with Certainty
Index

Factors 52, 6.3.1: Example 57, 6.4.1: Example 58

\textbf{def-\textit{kb-struct}} 70 6.2.1.1: KnowledgeWorks Structures 55

\textbf{def-\textit{named-kb-class}} 71 4.1.2: Named Classes 30

\textbf{defrule} 72 3: Rules 23, 3.1.2.1: Example 25, 3.2.2.1: Example 28, 6.1.1: Meta Rule Protocol 49, 6.1.1.4: Reasoning with Certainty Factors 52, 6.2.3.1: Pattern Matching 56, 6.2.3.2: Tail Recursion 57, 6.3.1: Example 58, 6.4.1: Example 58, 6.5.2: The Current Inferencing State 59

\textbf{defstruct} 6.2.1.1: KnowledgeWorks Structures 55

\textbf{deftactic} 73 6.1.2: User-definable Conflict Resolution 53

\textbf{undefined-context} 107 3.1.6.1: Defining Contexts 27

\textbf{undefined-rule} 108 3.1.3: Defining Forward Chaining Rules 25

\textbf{with-rule-actions} 109

\textbf{make-\textit{inferencing-state}} function 89 6.5.1: Creating and Maintaining Inferencing States 59

\textbf{make-instance} generic function 90 4.1: CLOS objects 30, 4.1.1: Unnamed Classes 30, 4.1.2: Named Classes 30, 6.2.1.1: KnowledgeWorks Structures 55, 6.4: Logical Dependencies and Truth Maintenance 58

\textbf{mea} conflict resolution tactic / function 91 3.1.5.3: Conflict Resolution 26

\textbf{~mea} conflict resolution tactic / function 92 3.1.5.3: Conflict Resolution 26, 6.2.2.2: Optimization of the Strategy 56

\textbf{member} A.12: Built-in Predicates 126

\textbf{menu button}

\textbf{creep} 2.6.2: Single-Stepping Rules 19, 5.8.1: Spy Windows 45

\textbf{leap} 2.6.2: Single-Stepping Rules 19, 5.8.1: Spy Windows 45

\textbf{menu item}

\textbf{browse} 2.4.3: Class Browser 15

\textbf{class browser} 2.4.3: Class Browser 15

\textbf{Classes} 2.4.3: Class Browser 16

\textbf{clear} 5.3: Clearing KnowledgeWorks 39

\textbf{context} 5.7: The Rule Browser 44

\textbf{FC History} 2.4.4: Forward Chaining History 17

\textbf{Inspect} 5.6: The Objects Browser 44

\textbf{Instantiations} 5.6: The Objects Browser 44

\textbf{KnowledgeWorks} 2.2: Loading the Tutorial 12, 2.4.2: Objects Browser 14, 2.4.3: Class Browser 16, 2.4.4: Forward Chaining History 17, 5.6: The Objects Browser 43, 5.7: The Rule Browser 44

\textbf{Listener} 2.2: Loading the Tutorial 12

\textbf{Objects} 2.4.2: Objects Browser 14, 5.6: The Objects Browser 43

\textbf{Rules} 2.4.1: Rule Browser 13, 5.7: The Rule Browser 44

\textbf{:meta} keyword 6.1.1: Meta Rule Protocol 49

\textbf{meta-interpreter} 6.1.1: Meta Rule Protocol 49

\textbf{Meta Object Protocol (MOP)} 1.1.2: Technical Overview 9

\textbf{metaprotocols} 1.1.2: Technical Overview 8

\textbf{meta-rule} 6.1.1: Meta Rule Protocol 49

\textbf{Meta Rule Protocol (MRP)} 1.1.2: Technical Overview 9, 6.1.1: Meta Rule Protocol 49

\textbf{mixin} 1.1.2: Technical Overview 9, 4.1.1: Unnamed Classes 30, 4.1.2: Named Classes 30

\textbf{monitor window} 2.6.1: Monitoring Forward Chaining Rules 18, 5.9: Monitor Windows 45

158
Index

MYCIN  6.1.1.4 : Reasoning with Certainty Factors  52

N
named classes  4.1.2 : Named Classes  30
named-kb-object class  93  4.1.2 : Named Classes  30
node  2.4.1 : Rule Browser  13
nodebug  A.12 : Built-in Predicates  126
no-debug function  94  3.1.7 : Forward Chaining Debugging  27
nonvar  A.12 : Built-in Predicates  126
nospy  A.12 : Built-in Predicates  126
not  3.1.2 : Forward Chaining Syntax  24,  A.12 : Built-in Predicates  126
not backward chaining goal  94
notrace  A.12 : Built-in Predicates  126

O
object  3.2.3 : Objects  28
  browser  2.6.2 : Single-Stepping Rules  19,  5.6 : The Objects Browser  42
  certainty factor  6.1.1.4 : Reasoning with Certainty Factors  52
  named  4.1 : CLOS objects  30
object base  2.6.1 : Monitoring Forward Chaining Rules  19,  4.3 : KnowledgeWorks Structures  36
  and inferencing states  6.5 : Inferencing States  59
  clearing  5.3 : Clearing KnowledgeWorks  39
  main chapter  4 : Objects  30
  uncertainty  6.1.1.4 : Reasoning with Certainty Factors  52,  6.1.1.4 : Reasoning with Certainty Factors  52
object browser  2.4.2 : Objects Browser  14
object system  1.1.2 : Technical Overview  8
once  A.12 : Built-in Predicates  127
OPS5  6.2.1.1 : KnowledgeWorks Structures  55,  E.1 : OPS5  144
optimization  6.2 : Optimization  54
optimization of KnowledgeWorks  4.3 : KnowledgeWorks Structures  36
order conflict resolution tactic / function  95  3.1.5.2 : Contexts  26,  3.1.5.3 : Conflict Resolution  26
~order conflict resolution tactic / function  96  3.1.5.3 : Conflict Resolution  26
output-defrels  A.12 : Built-in Predicates  127

P
pattern  2.4.2 : Objects Browser  15,  5.6 : The Objects Browser  43
  matching  6.2.3.1 : Pattern Matching  56
phrase  A.12 : Built-in Predicates  127
popup  2.3 : Running the Tutorial  12
Preferences... command  2.4.2 : Objects Browser  15,  5.6 : The Objects Browser  43,  5.7 : The Rule Browser  44
*print-verbose* variable  97
priority conflict resolution tactic / function  97  3.1.5.2 : Contexts  26,  3.1.5.3 : Conflict Resolution  26
Index

-priority conflict resolution tactic / function 98 3.1.5.3: Conflict Resolution 26
procedural language 1.1.2: Technical Overview 8
programming environment
main chapter 5: The Programming Environment 37
Prolog 3.2.1: Overview 27, 3.2.6: Edinburgh Prolog Translator 29, E.2: Prolog 146
accessing Lisp A.5: Accessing Lisp From Common Prolog 113
adding built in predicates A.13: Adding Built-in Predicates 128
built in predicates A.12: Built-in Predicates 125
calling from LispWorks A.6: Calling Prolog From Lisp 114
cut 6.2.3.3: Cut 57
debugging A.7: Debugging 118
Edinburgh Syntax A.10: Edinburgh Syntax 124
exiting the interpreter A.4.4: Exiting the Interpreter 113
graphic environment A.11: Graphic Development Environment 125
interface functions A.6.2: Interface Functions 116
leashing A.7: Debugging 118
logic interpreter A.4: Using The Logic Interpreter 111
logic listener A.11: Graphic Development Environment 125
macros A.8: Common Prolog Macros 122
main chapter Appendix A: Common Prolog 110
overview A.1.1: Overview 110
predicates compatible with Edinburgh syntax A.14: Edinburgh Compatibility Predicates 129
retrieving multiple solutions in A.4.1: Multiple Solutions 112
specifying multiple goals in A.4.2: Multiple Goals 112
spy points A.7: Debugging 118
syntax A.2: Syntax 110
tracing A.7: Debugging 118

Q
query 5.6: The Objects Browser 43

R
read-query-print loop A.4: Using The Logic Interpreter 111
read-term A.12: Built-in Predicates 127
recency conflict resolution tactic / function 99 3.1.5.2: Contexts 26, 3.1.5.3: Conflict Resolution 26
-recency conflict resolution tactic / function 100 3.1.5.3: Conflict Resolution 26, 6.2.2.2: Optimization of the Strategy 56
record A.12: Built-in Predicates 127
recorded A.12: Built-in Predicates 127
recordz A.12: Built-in Predicates 127
relational database classes A.2: Relational Database Objects 31
repeat A.12: Built-in Predicates 127
reset function 101 5.3: Clearing KnowledgeWorks 39, 6.5.2: The Current Inferencing State 59

160
Index

**retract** A.12: Built-in Predicates 127

**return** 3.1.2: Forward Chaining Syntax 24, 3.1.5.2: Contexts 26

**return** backward chaining goal 101

**rule** 3: Rules 23

  action 3.1.1: Overview 23

  backward chaining 3.2.4: Defining Backward Chaining Rules 28

  browser 2.6.1: Monitoring Forward Chaining Rules 18

  condition 3.1.1: Overview 23

  definition of forward chaining 3.1.3: Defining Forward Chaining Rules 25

  editing definitions 2.6.3: Editing Rule Definitions 19

  groups 2.4.4: Forward Chaining History 17

  implication strength 6.1.1.4: Reasoning with Certainty Factors 52

  single-stepping 2.6.2: Single-Stepping Rules 19

**rulebase** 4.2: Relational Database Objects 31

**rule browser** 5.7: The Rule Browser 44

**rule-defined conflict resolution** 6.2.2.2: Optimization of the Strategy 56

**rule monitor** 5.7: The Rule Browser 44, 5.9: Monitor Windows 45

**rule preconditions** 6.2.1.2: Efficient Forward Chaining Rule Preconditions 55

Rules

  main chapter 3: Rules 23

**S**

**setof** A.12: Built-in Predicates 127

**signal-4b-name-clash** variable 102

**sort** A.12: Built-in Predicates 127

**specificity** conflict resolution tactic / function 102 3.1.5.3: Conflict Resolution 26

**-specificity** conflict resolution tactic / function 103 3.1.5.3: Conflict Resolution 26

**spy** A.12: Built-in Predicates 127

spy points A.7: Debugging 118

spy window 2.6.2: Single-Stepping Rules 19, 5.7: The Rule Browser 44, 5.8.1: Spy Windows 45

**standard-class** class 6.3: Use of Meta-Classes 57

**standard-context** backward chaining goal 104

**standard-db-object** class 4.2.1: Example 31

**standard-kb-class** 6.3.1: Example 57

**standard-kb-object** class 105 4.2.1: Example 31, 5.5: The Class Browser 40, 6.3: Use of Meta-Classes 57, def-kb-class 69

**start-cycle** backward chaining goal 105 6.1.1: Meta Rule Protocol 49

**start-kw** function 106

static conflict resolution 6.2.2.2: Optimization of the Strategy 56

structures 4.3: KnowledgeWorks Structures 36, 5.3: Clearing KnowledgeWorks 39, 6.2.1.1: KnowledgeWorks Structures 54

subclasses 2.4.3: Class Browser 16
Index

symbol macros
  *cycle*  67

syntax
  backward chaining  3.2.2: Backward Chaining Syntax  28
  expression  3.1.2: Forward Chaining Syntax  24
  forward-condition  3.1.2: Forward Chaining Syntax  23
  of forward chaining  3.1.2: Forward Chaining Syntax  23
  syntax of Prolog  A.2: Syntax  110
  system browser  5.4: The System Browser  39
  systems  2.8: Systems  21

T
tactic  3.1.5.3: Conflict Resolution  26,  deftactic  73
Tail Recursion  6.2.3.2: Tail Recursion  57
test  backward chaining goal  107

Tools menu
  Preferences...  2.4.2: Objects Browser  15,  5.6: The Objects Browser  43,  5.7: The Rule Browser  44
  trace  A.12: Built-in Predicates  127
    in Prolog  A.7: Debugging  118
  translate-vars  A.12: Built-in Predicates  127
  true  A.12: Built-in Predicates  127

truth maintenance  3.1.2: Forward Chaining Syntax  24,  6.4: Logical Dependencies and Truth Maintenance  58

Tutorial
  main chapter  2: Tutorial  11

U
undefcontext macro  107  3.1.6.1: Defining Contexts  27
undefrule macro  108  3.1.3: Defining Forward Chaining Rules  25

unnamed classes  4.1.1: Unnamed Classes  30

V
validate-superclass generic function  6.3: Use of Meta-Classes  57

var A.12: Built-in Predicates  127

variables
  *inferencing-state*  80  6.5.2: The Current Inferencing State  59
  *in-interpreter*  82
  *print-verbose*  97
  *signal-kb-name-clash*  102

W
window

  browser  2.4: Browsers  13

  editor  2.4.1: Rule Browser  13,  5.2: The Editor  38

162
Index

listener 2.2: Loading the Tutorial 12, 2.3: Running the Tutorial 12, 5.1: The KnowledgeWorks Listener 37
monitor 2.6.1: Monitoring Forward Chaining Rules 18, 5.9: Monitor Windows 45
podium 2.1: Getting Started 11
popup 2.3: Running the Tutorial 12
spy 2.6.2: Single-Stepping Rules 19, 5.7: The Rule Browser 44, 5.8.1: Spy Windows 45
subclasses 2.4.3: Class Browser 16

with-prolog A.6.2.3: with-prolog 117
with-rule-actions macro 109

Non-alphanumerics

* 1.2.1: Prolog syntax 10
+ 1.2.1: Prolog syntax 10
/== A.12: Built-in Predicates 125
<..> 1.2.1: Prolog syntax 10
= A.12: Built-in Predicates 125
=.. A.2: Syntax 111, A.12: Built-in Predicates 125
::= 1.2.1: Prolog syntax 10
== A.12: Built-in Predicates 125
@< A.12: Built-in Predicates 125
@=< A.12: Built-in Predicates 125
@> A.12: Built-in Predicates 125
@>= A.12: Built-in Predicates 125
[..] 1.2.1: Prolog syntax 10
| 1.2.1: Prolog syntax 10