KnowledgeWorks® and Prolog User Guide

Version 6.1

Copyright and Trademarks

KnowledgeWorks and Prolog User Guide (Macintosh version)

Version 6.1

December 2011

Address

LispWorks Ltd
St. John’s Innovation Centre
Cowley Road
Cambridge
CB4 0WS
England

Telephone

From North America: 877 759 8839
(toll-free)

From elsewhere: +44 1223 421860

Fax

From North America: 617 812 8283

From elsewhere: +44 870 2206189

www.lispworks.com

Contents

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. The following sections provide an historical perspective and an overview of the system.

1.2 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.

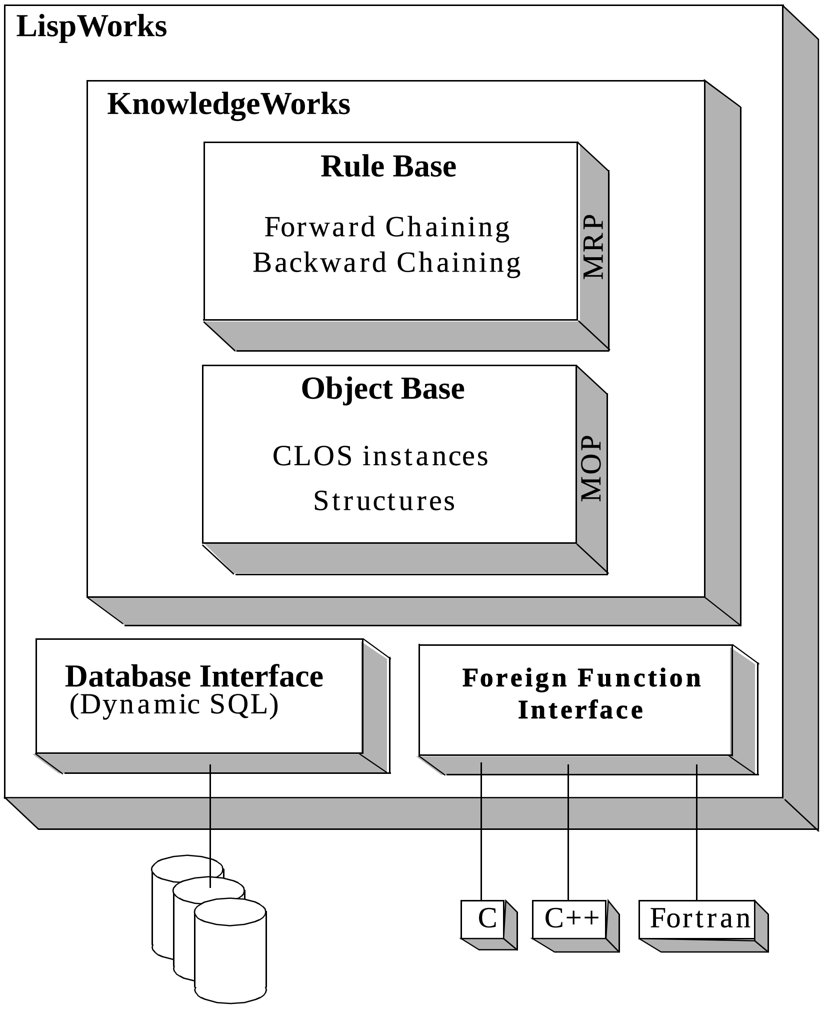
1.3 Technical Overview

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



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 *LispWorks Editor User Guide*).

1.3.1 Appearance of the graphical tools

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.

1.4 Notation Conventions

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

2

Tutorial

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 **use**s the **KW** package, for instance, **KW-USER**.

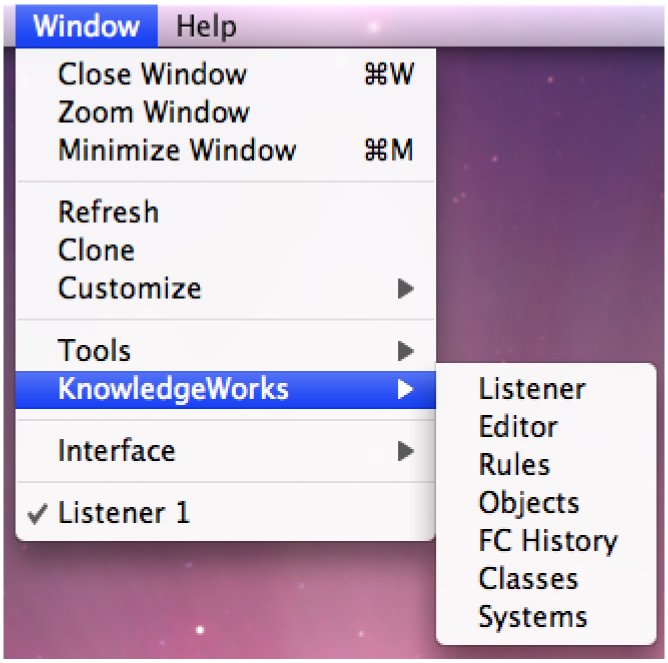
2.1 Getting Started

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

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

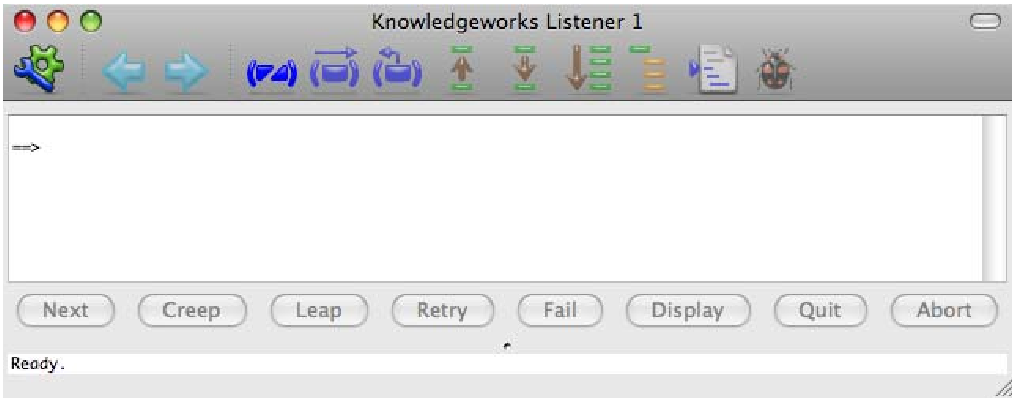
Start LispWorks. The LispWorks menu bar and the LispWorks toolbar will appear. Note the position of the **KnowledgeWorks** menu, which you will use to access the tools described in this manual.

Figure 2.1 KnowledgeWorks menu



2.2 Loading the Tutorial

Figure 2.2 KnowledgeWorks Listener



First bring up a KnowledgeWorks Listener by choosing **Window > Knowledge-Works > Listener** from the LispWorks menu bar. 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 (system:lispworks-dir "examples/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)
```

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:

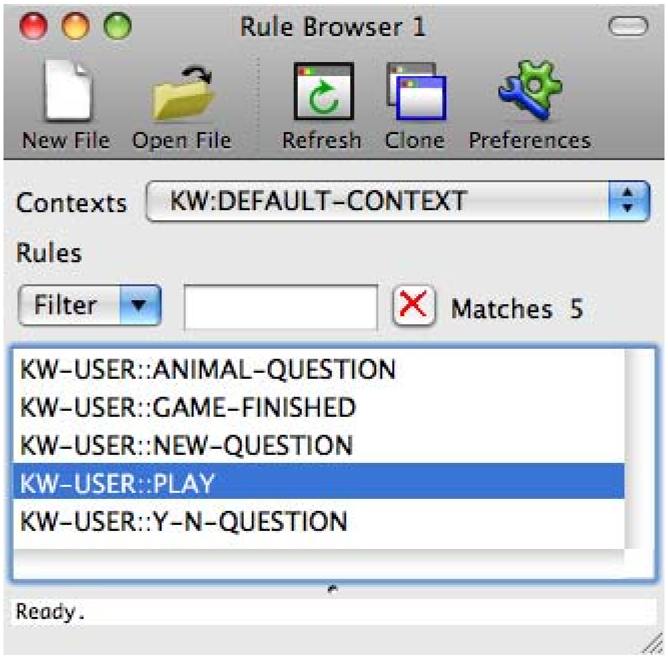
1. Another popup will ask what the animal was. Type in the name of an animal and press `Return` (or click on **OK**). If the animal is already known to the system this constitutes an error. A confirmer popup will inform you of this; click on Confirm and execution will terminate.
2. You will be asked 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 `(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 Chapter 5, "The Programming Environment".

2.4.1 Rule Browser

Figure 2.3 KnowledgeWorks Rule Browser



KW:DEFAULT-CONTEXT

KW-USER::ANIMAL-QUESTION
KW-USER::GAME-FINISHED
KW-USER::NEW-QUESTION
KW-USER::PLAY
KW-USER::Y-N-QUESTION

Ready.

This may be obtained by choosing **Window > 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 **DEFAULT-CONTEXT**. Below that are listed the rules for the selected context. Choose **DEFAULT-CONTEXT** from the drop-down list and click on one of the rules, for example **PLAY**, and edit it by choosing **Rule > Find Source** from the menu bar. An editor window will appear showing this rule definition.

What this rule says is:

```
(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))
```

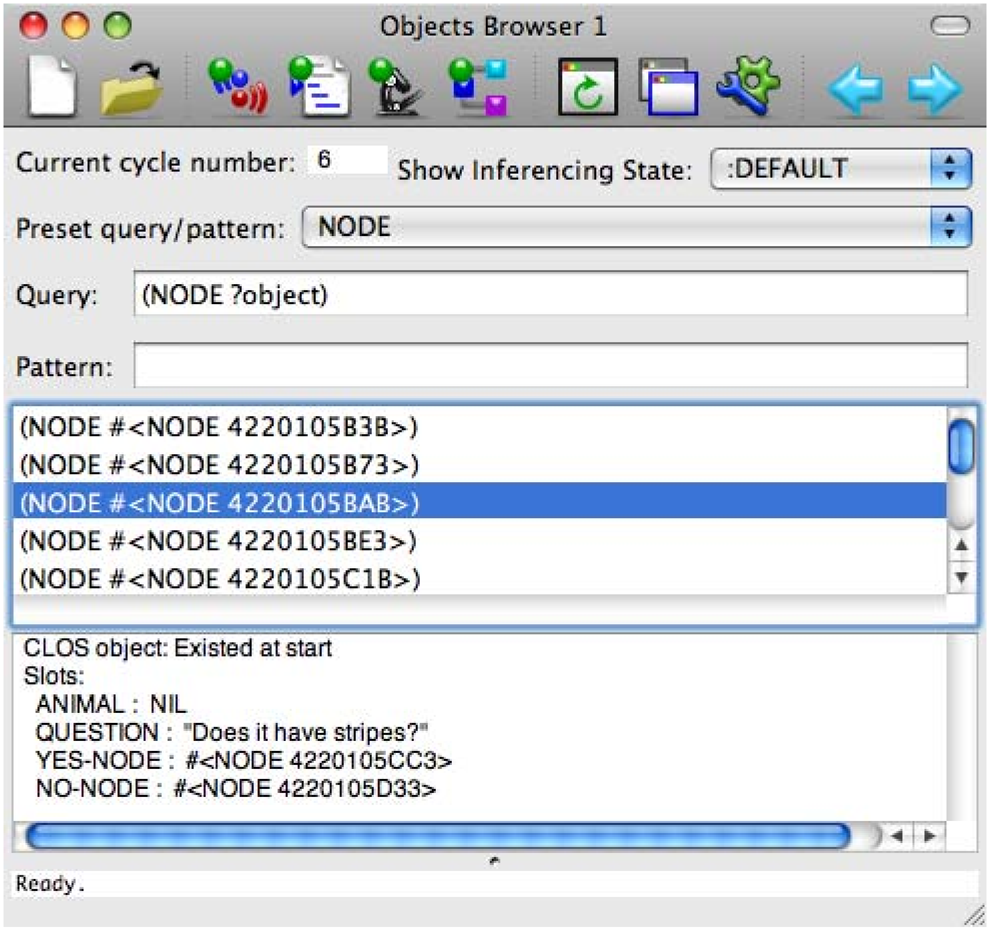
which means:

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 top-most question in the tree of questions”. The detailed syntax of forward chaining rule definitions will be explained in Chapter 3, “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 Chapter 3, “Backward Chaining”.

2.4.2 Objects Browser

Figure 2.4 KnowledgeWorks Objects Browser



Current cycle number: 6 Show Inferencing State: :DEFAULT

Preset query/pattern: NODE

Query: (NODE ?object)

Pattern:

```
(NODE #<NODE 4220105B3B>)
(NODE #<NODE 4220105B73>)
(NODE #<NODE 4220105BAB>)
(NODE #<NODE 4220105BE3>)
(NODE #<NODE 4220105C1B>)
```

```
CLOS object: Existed at start
Slots:
 ANIMAL : NIL
 QUESTION : "Does it have stripes?"
 YES-NODE : #<NODE 4220105CC3>
 NO-NODE : #<NODE 4220105D33>
```

Ready.

The Objects Browser is for exploring the contents of the KnowledgeWorks object base. Start it by choosing **Window > 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 and slot values of the object.

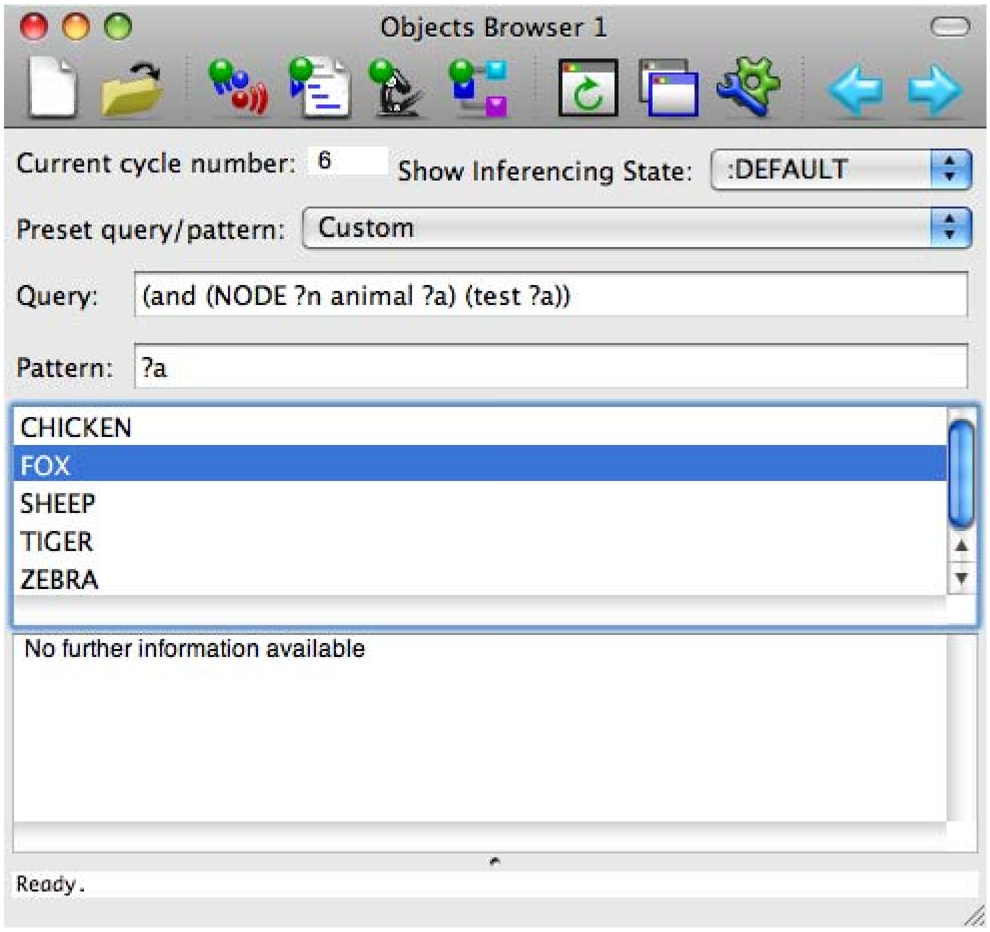
To make the display clearer and allow input without explicit package qualifiers, change the package of the Objects Browser. Do this via **LispWorks > Preferences... > Objects Browser > Package**. Edit the Package pane so that it says `KW-USER` and press **OK**.

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`.

Figure 2.5 Objects Browser matching animals



Current cycle number: 6 Show Inferencing State: :DEFAULT

Preset query/pattern: Custom

Query: (and (NODE ?n animal ?a) (test ?a))

Pattern: ?a

```
CHICKEN
FOX
SHEEP
TIGER
ZEBRA
```

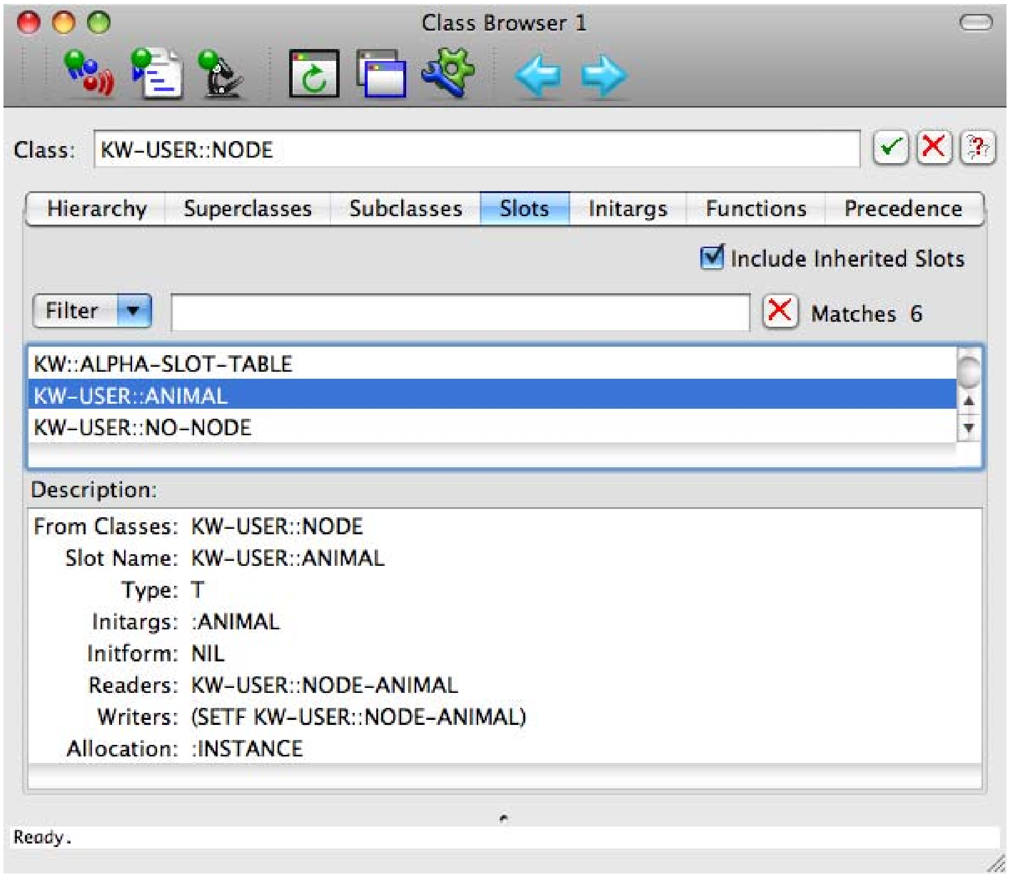
No further information available

Ready.

Only the non-nil animals are displayed.

2.4.3 Class Browser

Figure 2.6 KnowledgeWorks Class Browser



KW::ALPHA-SLOT-TABLE
KW-USER::ANIMAL
KW-USER::NO-NODE

Description:

From Classes:	KW-USER::NODE
Slot Name:	KW-USER::ANIMAL
Type:	T
Initargs:	:ANIMAL
Initform:	NIL
Readers:	KW-USER::NODE-ANIMAL
Writers:	(SETF KW-USER::NODE-ANIMAL)
Allocation:	:INSTANCE

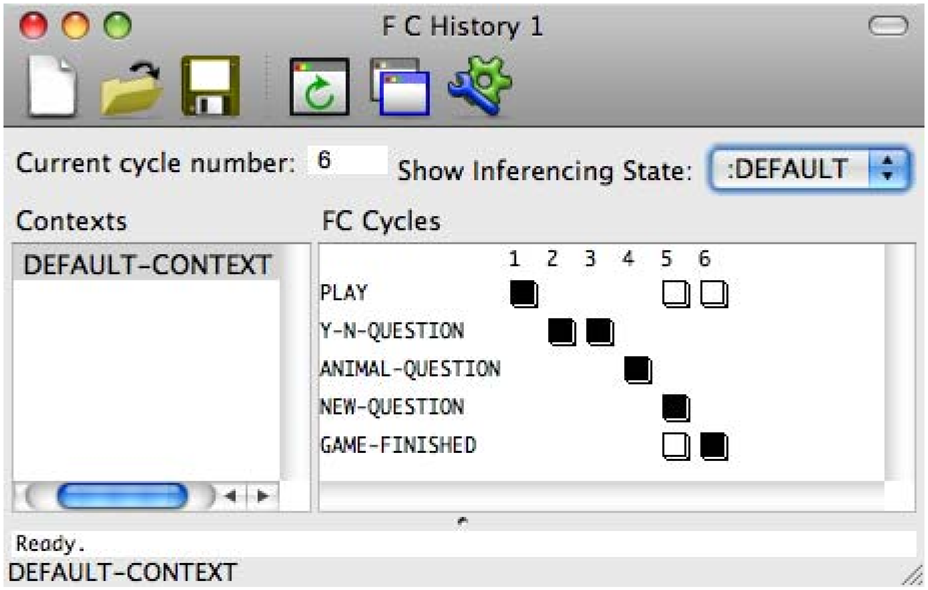
Ready.

The Class Browser is obtained by choosing **Window > 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

Figure 2.7 KnowledgeWorks Forward Chaining History



F C History 1

Current cycle number: 6 Show Inferencing State: :DEFAULT

Contexts

DEFAULT-CONTEXT

FC Cycles

	1	2	3	4	5	6
PLAY	■				□	□
Y-N-QUESTION		■	■			
ANIMAL-QUESTION				■		
NEW-QUESTION					■	
GAME-FINISHED					□	■

Ready.

DEFAULT-CONTEXT

This is obtained by choosing **Window > 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 "Objects Browser" on page 13).

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

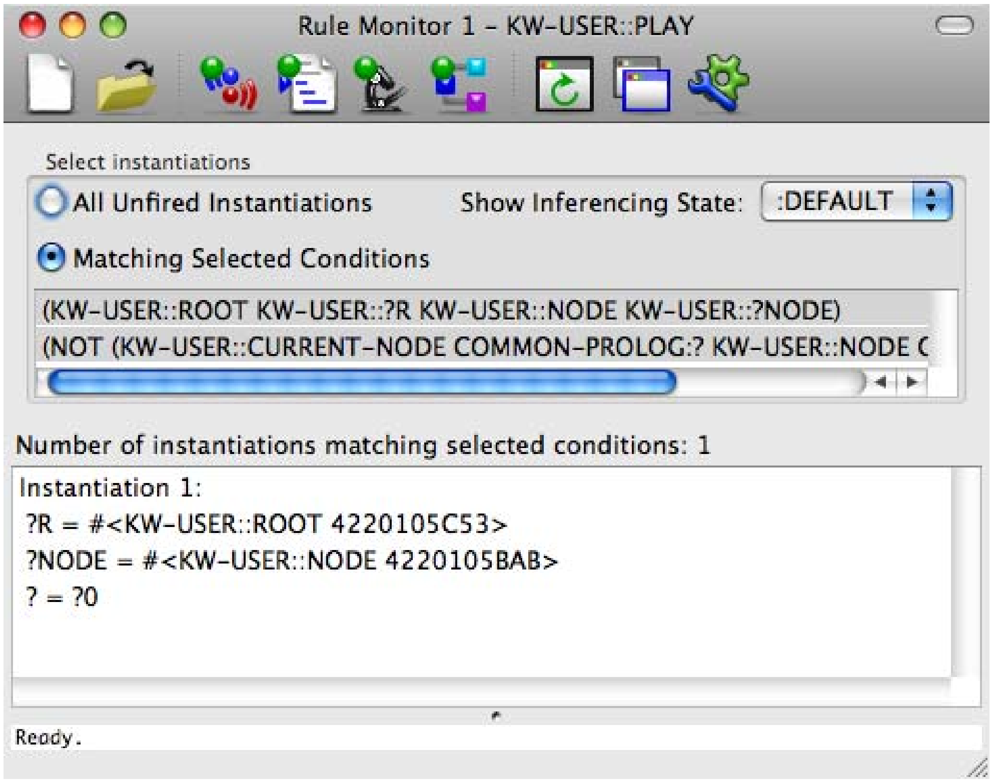
The KnowledgeWorks Listener has already been shown to function as a Lisp Listener. However it extends this with the ability of the Objects Browser to match objects. When using the Objects Browser the **Query** pane contained patterns which could be matched against the Object Base. These same patterns 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

Figure 2.8 KnowledgeWorks Rule Monitor


Rule Monitor 1 – KW-USER::PLAY
Select instantiations
All Unfired Instantiations
Matching Selected Conditions
Show Inferencing State:
:DEFAULT

(KW-USER::ROOT KW-USER::?R KW-USER::NODE KW-USER::?NODE)
(NOT (KW-USER::CURRENT-NODE COMMON-PROLOG:? KW-USER::NODE (

Number of instantiations matching selected conditions: 1

```
Instantiation 1:
 ?R = #<KW-USER::ROOT 4220105C53>
 ?NODE = #<KW-USER::NODE 4220105BAB>
 ? = ?0
```

Ready.

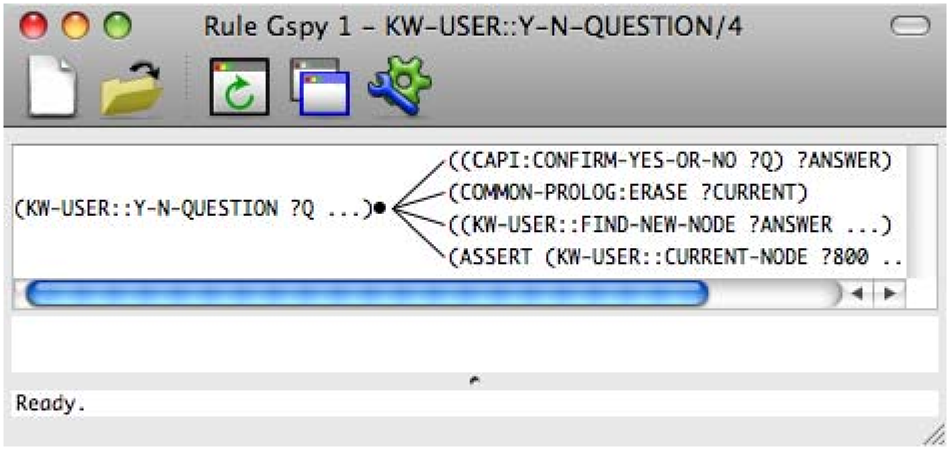
One of the problems with forward chaining rules is determining why they are (or are not) being matched. To deal with this KnowledgeWorks has Monitor Windows for forward chaining rules. To bring up a Monitor Window, select the `DEFAULT-CONTEXT` in the Rule Browser, click on `PLAY` and choose **Rule > 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 **Window > Refresh** from the Rule Monitor menu bar, or **Memory > Update Monitor Windows** from the KnowledgeWorks Listener. When you are single-stepping through rules (see below) Monitor windows are updated automatically.

2.6.2 Single-Stepping Rules

Figure 2.9 KnowledgeWorks Gspy Window



Rule Gspy 1 – KW-USER::Y-N-QUESTION/4

```
(KW-USER::Y-N-QUESTION ?Q ...)
    ((CAPI:CONFIRM-YES-OR-NO ?Q) ?ANSWER)
    (COMMON-PROLOG:ERASE ?CURRENT)
    ((KW-USER::FIND-NEW-NODE ?ANSWER ...)
    (ASSERT (KW-USER::CURRENT-NODE ?800 ..
```

Ready.

Select a rule, say, `Y-N-QUESTION`, in the Rule Browser and choose **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.

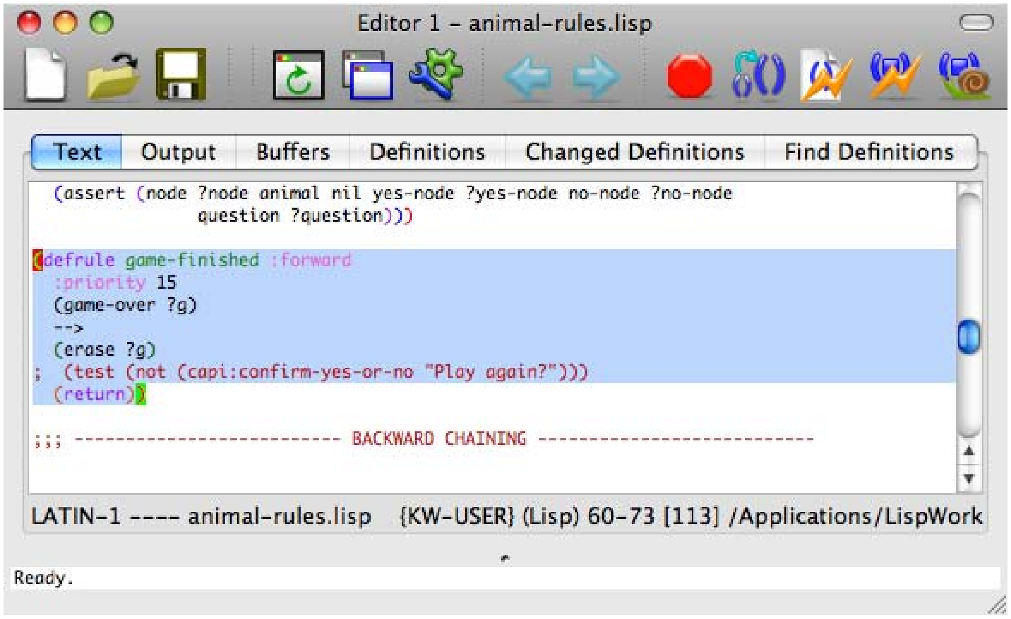
Now enter `(infer)` in the Listener to run the demo again. Execution will stop when this rule fires. A message in the listener will say that the rule **`Y-N-QUESTION`** has been called. Click on the **Creep** button at the bottom of the Listener to single step through the rule. Watch the highlight move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

Click on **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 **Window > Close Window**) and press **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 **Rule > Gspy**).

2.6.3 Editing Rule Definitions

Figure 2.10 KnowledgeWorks Editor



```
  (assert (node ?node animal nil yes-node ?yes-node no-node ?no-node
                question ?question)))

(defrule game-finished :forward
  :priority 15
  (game-over ?g)
  -->
  (erase ?g)
;   (test (not (capi:confirm-yes-or-no "Play again?")))
  (return))

;;; ------------------------- BACKWARD CHAINING --------------------------
```

```
LATIN-1 ---- animal-rules.lisp    {KW-USER} (Lisp) 60-73 [113] /Applications/LispWork
```

```
Ready.
```

Let us suppose that when the demo finishes we would like it to ask if we want to play again. Find the definition for **`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 **Definitions > Compile** from the editor menu bar. Press **`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 **`(return)`** instruction ends execution) if requested. Run the demo to see this happen.

The rule **`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.

2.7 Lisp Integration

You can save your object base of animals by entering

```
(save-animals "my-animal-objs.lisp")
```

into the Listener. In the file of rules `"animal-rules.lisp"` look at the function `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.

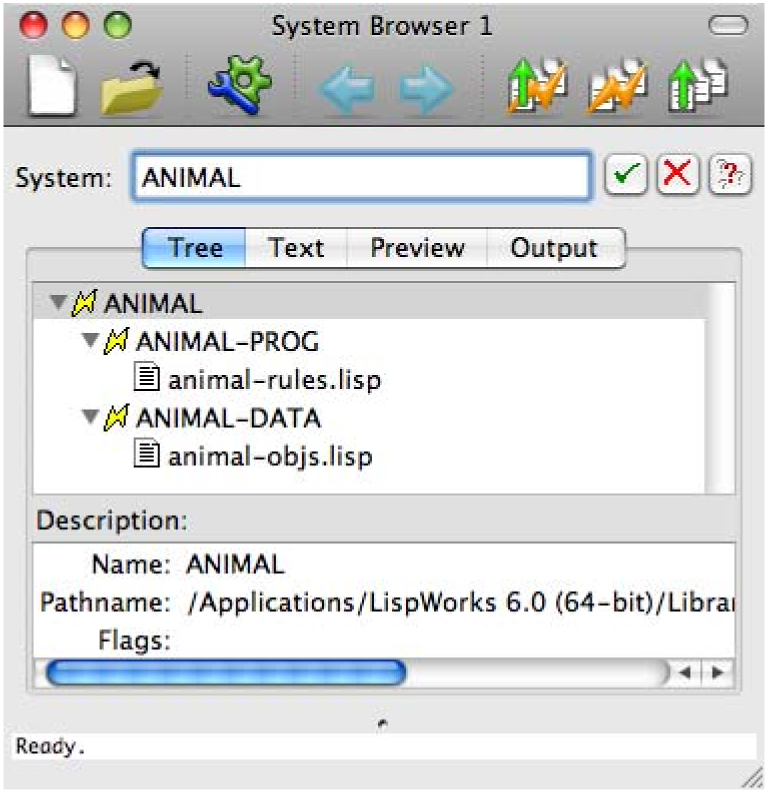
Note: KnowledgeWorks CLOS objects are ordinary CLOS objects and can be used outside KnowledgeWorks rules.

2.7.1 The LispWorks IDE

The entire programming environment of the LispWorks IDE is available from the menus on the LispWorks menu bar and the LispWorks toolbar. See the *LispWorks IDE User Guide* for more details.

2.8 Systems

Figure 2.11 KnowledgeWorks System Browser



System: ANIMAL

ANIMAL
ANIMAL-PROG
animal-rules.lisp
ANIMAL-DATA
animal-objs.lisp

Name: ANIMAL
Pathname: /Applications/LispWorks 6.0 (64-bit)/Libra
Flags:

If you are familiar with LispWorks system definitions, look at the system definition for the animal demo in the Editor tool by choosing **File > Open...** and navigating to the file `examples/kw/animal/defsystem.lisp`. It contains a `KB-SYSTEM` and a `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 `Esc X Describe System` or **File > Browse Parent System**.

`KB-SYSTEM`s are reloaded when the rules are cleared. `KB-INIT-SYSTEM`s 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 **Window > Close Window**. You can exit LispWorks by choosing **LispWorks > Quit LispWorks**. 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 Section 3.1.5, “Control Flow”) defaulting to **`default-context`**, and **`<priority-number>`** is a number (see Section 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>]*)

<object-slot-condition> ::=
     <slot-name> <term>
```

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

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

(test <lisp-expr>) is a Lisp test where **<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 **<object-slot-condition>** syntax. They cannot depend on values obtained by calling **slot-value** or a reader function.

(not <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.

(logical <forward-condition>+) is used to indicate clauses that describe the *logical dependencies* amongst objects. See Section 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

```
(<class-name> <variable> ...)
```

The syntax for expressions is:

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

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

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

(assert (<class-name> <variable> [<slot-name> <term>]*)) is an assertion which modifies the contents of the object base, where if **<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.

(context <context-list>) adds the given list of contexts to the top of agenda (see Section 3.1.5, “Control Flow”).

(return) passes control to the top context on the agenda and removes it from the agenda (see Section 3.1.5, “Control Flow”).

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

<goal> may be any backward chaining goal expression (see Section 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.

3.1.2.1 Example

```
(defrule move-train :forward
         :context train
         (train ?train position ?train-pos)
         (signal ?signal position ?signal-pos
          colour green)
         (test (= ?signal-pos (1+ ?train-pos)))
-->
  ((format t "~%Train moving to position ~s"
            ?signal-pos))
  (assert (signal ?signal colour 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:

```
<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 Section 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 favours 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` – (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` – (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, Farrel, 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 Section 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 Chapter 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 Knowledge-Works 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.

3.2.2 Backward Chaining Syntax

Backward chaining rule bodies are defined as:

```
<body> ::= <clause>+
<clause> ::= (<goal> <-- <expression>*)
<goal> ::= (<rule-name> <term>*)
```

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

3.2.2.1 Example

```
(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)))
```

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 **`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 **`(<class-name> <variable> [<slot-name> <term>]*)`** syntax, and these expressions are instantiated directly without creating any sub-goals. The **`<class-name>`** of any CLOS class or KnowledgeWorks structure may not coincide with any backward chaining **`<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

```
(any <expr-to-instantiate> <expr-to-prove>)
```

which finds any solution to **`<expr-to-prove>`** and instantiates **`<expr-to-instantiate>`**, and

```
(findall <expr-to-instantiate> <expr-to-prove>)
```

finds all the solutions to **`<expr-to-prove>`**, instantiates **`<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”.

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

```
(any '(?x is in (1 2 3)) '(member ?x (1 2 3)))
```

returns

```
(1 is in (1 2 3))
```

The following expression:

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

returns

```
((1 is in (1 2 3))(2 is in (1 2 3))(3 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 Section 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. Chapter 5, “The Programming Environment”, explains this in detail. The leashing of the ports can be adjusted, details are to be found in Section A.7, “Debugging”.

3.3 Common Lisp Interface

Arbitrary Lisp expressions may be called from rules. See Section 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 Section 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

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

retrieves the instance from its name. The function

```
(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:

```
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 analyses 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
          :column price)
   (owner-name :type (string 20)
               :db-kind :base
               :column owner)
   (owner :accessor vehicle-owner
          :db-kind :join
          :db-info (:home-key owner-name
                    :foreign-key name
                    :join-class person
                    :set nil
                    :retrieval :deferred))))

;;; the person class maps onto the person table in the
;;; database
;;; vehicle is a join slot which looks up the owned
;;; vehicle object
;;; company is a join slot which looks up the company
;;; object
```

```
(sql:def-view-class person
     (sql: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))
   (employer :type (string 20)
             :db-kind :base
             :column employer)
   (company :accessor person-company
            :db-kind :join
            :db-info (:home-key employer
                      :foreign-key name
                      :join-class company
                      :set nil
                      :retrieval :deferred))))

;;; the company class maps onto the company table in
;;; the database

(sql:def-view-class company
       (sql: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 initialise 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 hasn't 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)
  (vehicles-for-person ?c-f-p
                       person ?person
                       vehicles ?vehicles)
  (test (not (member ?vehicle ?vehicles)))
        ; has it been included?
  -->
  (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 jag.
;;; 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)
                   ((?person . ?vehicle) . ?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 = ''))
(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
26
```

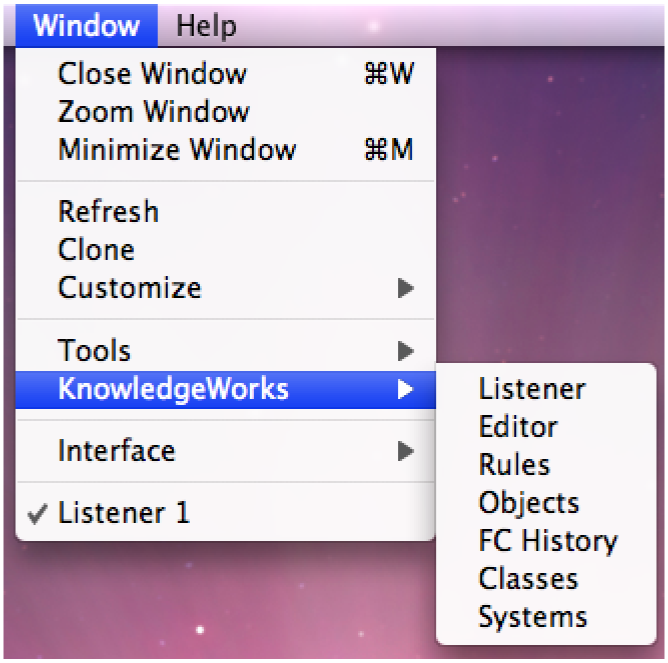
4.3 KnowledgeWorks Structures

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 Section 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.

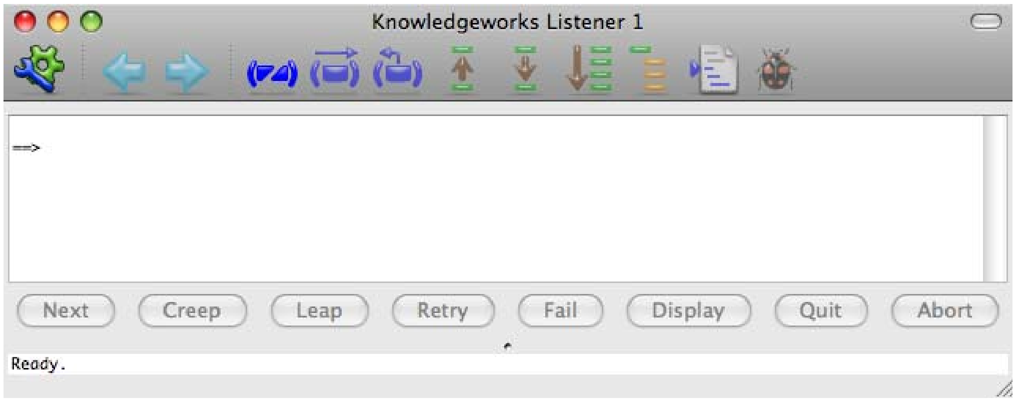
Figure 5.1 KnowledgeWorks Menu



All KnowledgeWorks windows can be closed independently of the others by choosing **Window > Close Window**. You can switch between windows by choosing **Window >** *window-name*.

5.1 The KnowledgeWorks Listener

Figure 5.2 KnowledgeWorks Listener



The KnowledgeWorks Listener is obtained by choosing **Window > 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

```
<expression>
```

as defined in Section 3.1, “Forward chaining”, or

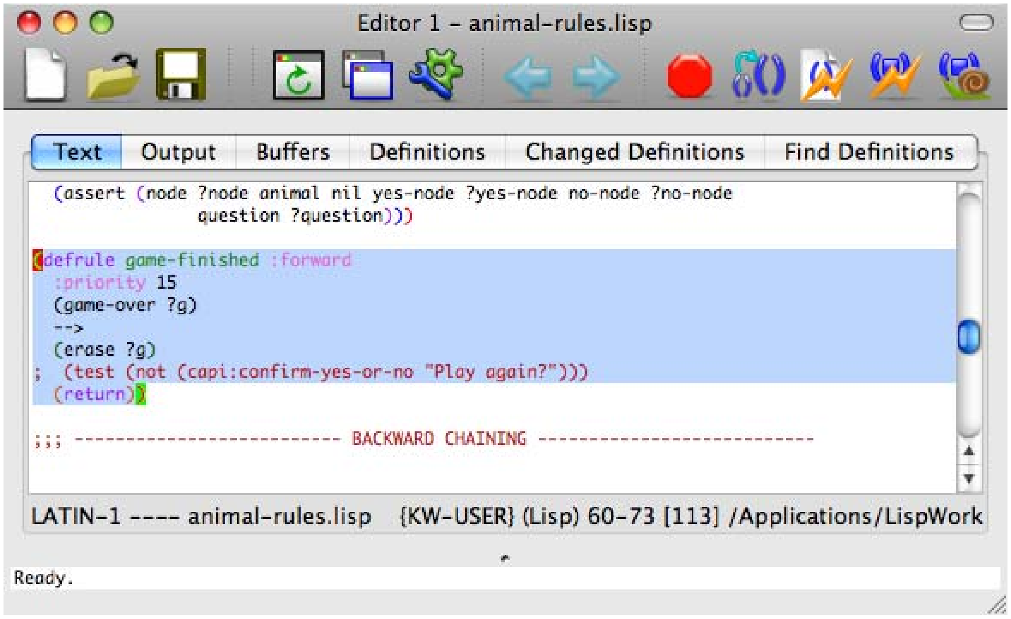
```
<lisp-expr>
```

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

Figure 5.3 KnowledgeWorks Editor



```
 (assert (node ?node animal nil yes-node ?yes-node no-node ?no-node
                question ?question)))

(defrule game-finished :forward
  :priority 15
  (game-over ?g)
  -->
  (erase ?g)
;  (test (not (capi:confirm-yes-or-no "Play again?")))
  (return))

;;; ------------------------- BACKWARD CHAINING --------------------------
```

```
LATIN-1 ---- animal-rules.lisp   {KW-USER} (Lisp) 60-73 [113] /Applications/LispWork
Ready.
```

The KnowledgeWorks Editor is created by choosing **Window > Knowledge-Works > Editor**. It is the same as the LispWorks Editor tool. Please see the *Lisp-Works IDE User Guide* for more information on the editor tool and the *LispWorks Editor User Guide* for information on the various editing commands.

5.3 Clearing KnowledgeWorks

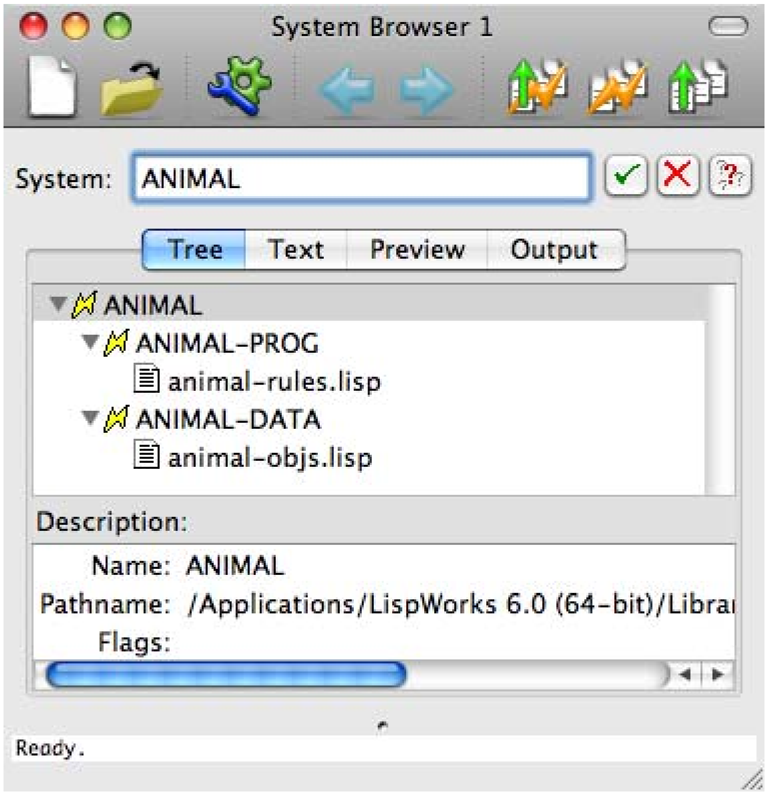
The KnowledgeWorks object base (all the KnowledgeWorks CLOS objects and any optimized structures) may be cleared by choosing **Memory > Clear Objects** from the KnowledgeWorks Listener, or by calling the function `reset`.

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

Figure 5.4 KnowledgeWorks System Browser



System: ANIMAL

```
▼ ANIMAL
  ▼ ANIMAL-PROG
      animal-rules.lisp
  ▼ ANIMAL-DATA
      animal-objs.lisp
```

Description:

```
    Name:  ANIMAL
Pathname:  /Applications/LispWorks 6.0 (64-bit)/Librar
   Flags:
```

Ready.

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

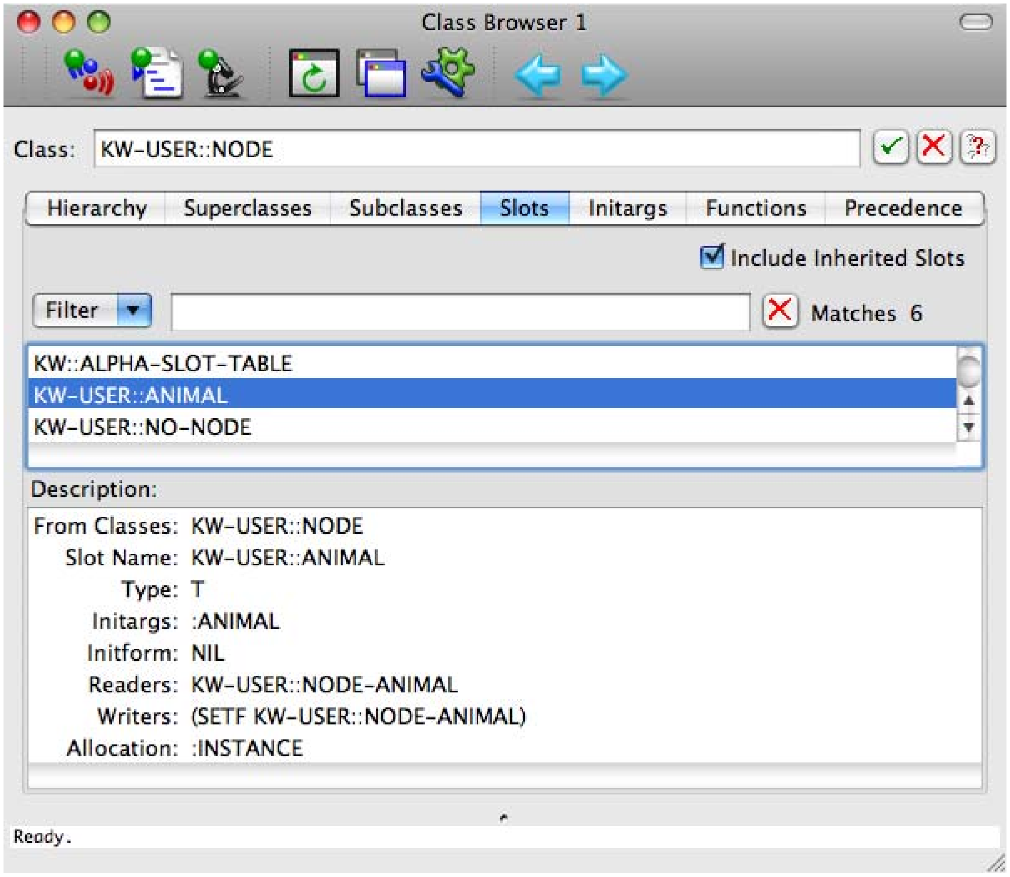
- `:kb-system`, which are reloaded when the KnowledgeWorks rules are cleared (see Section 5.3 on page 54).

- **`:kb-init-system`**, which are reloaded when the KnowledgeWorks object base is cleared (see Section 5.3 on page 54).

For more information on LispWorks systems, see the Common Defsystem chapter in the *LispWorks User Guide and Reference Manual*. For more information about the System Browser tool, see the *LispWorks IDE User Guide*.

5.5 The Class Browser

Figure 5.5 KnowledgeWorks Class Browser



Class: KW-USER::NODE

[✓ Include Inherited Slots]

Matches 6

KW::ALPHA-SLOT-TABLE
KW-USER::ANIMAL
KW-USER::NO-NODE

Description:

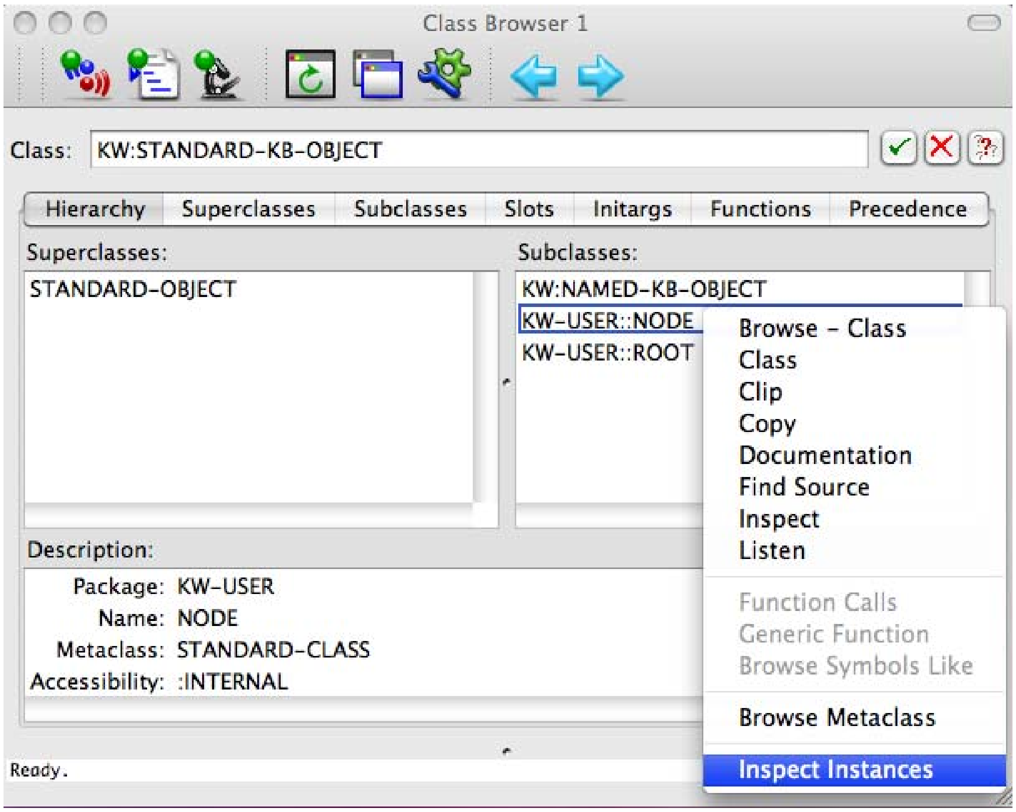
```
From Classes:  KW-USER::NODE
   Slot Name:  KW-USER::ANIMAL
        Type:  T
    Initargs:  :ANIMAL
    Initform:  NIL
     Readers:  KW-USER::NODE-ANIMAL
     Writers:  (SETF KW-USER::NODE-ANIMAL)
  Allocation:  :INSTANCE
```

Ready.

The Class Browser is obtained by choosing **Window > KnowledgeWorks > Classes**. It is the same as the LispWorks Class Browser except that

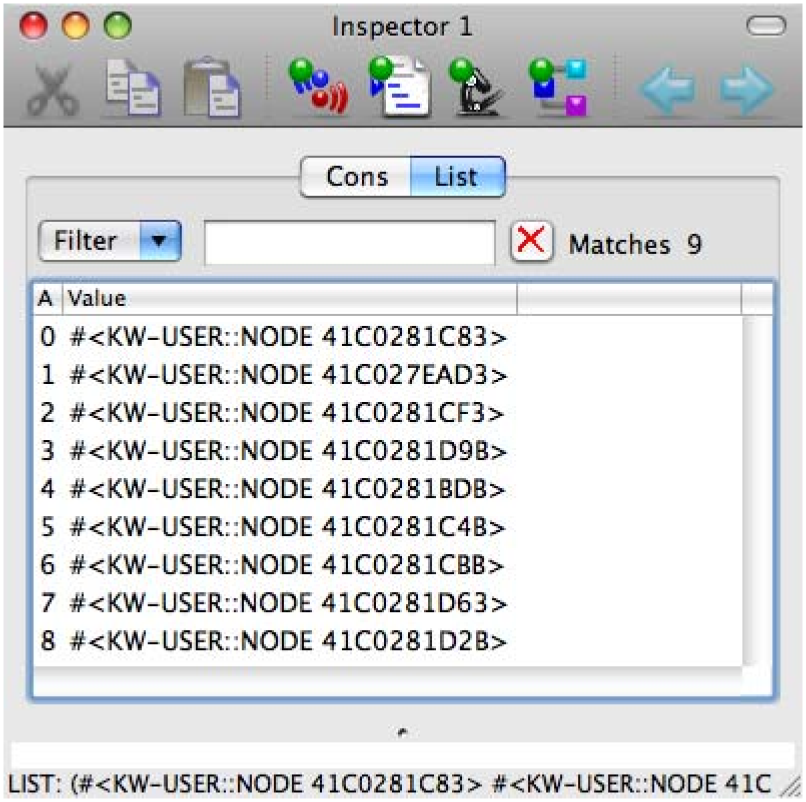
- it appears with an initial focus on `standard-kb-object`
- when looking at a KnowledgeWorks class the **Classes** menu and context menu contain an **Inspect Instances** command which allows you to look at the instances of the class.

Figure 5.6 Inspecting instances from the Class Browser



This raises an Inspector tool with a list of all the instances.

Figure 5.7 KnowledgeWorks Instances Inspector



```
0 #<KW-USER::NODE 41C0281C83>
1 #<KW-USER::NODE 41C027EAD3>
2 #<KW-USER::NODE 41C0281CF3>
3 #<KW-USER::NODE 41C0281D9B>
4 #<KW-USER::NODE 41C0281BDB>
5 #<KW-USER::NODE 41C0281C4B>
6 #<KW-USER::NODE 41C0281CBB>
7 #<KW-USER::NODE 41C0281D63>
8 #<KW-USER::NODE 41C0281D2B>
```

```
LIST: (#<KW-USER::NODE 41C0281C83> #<KW-USER::NODE 41C
```

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:

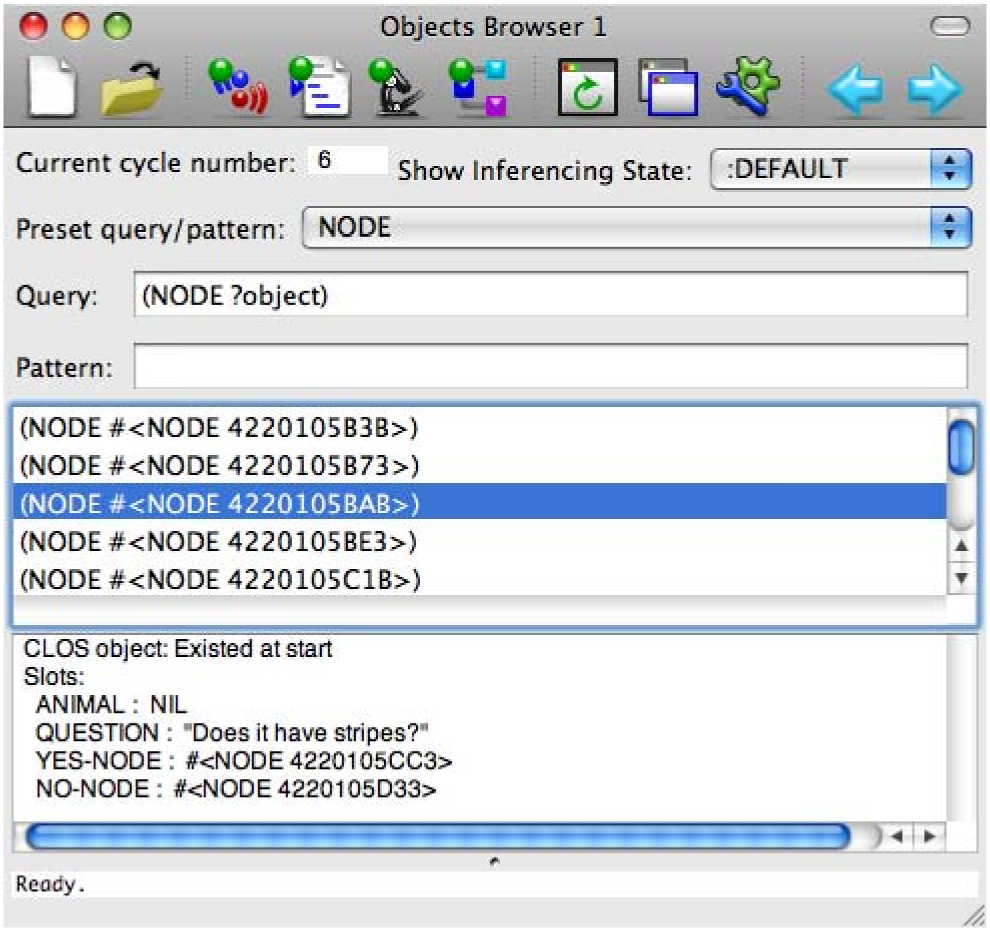
- **Superclasses** and **Subclasses** tabs to draw a graphs of the superclasses or subclasses of the class being looked at
- **Slots** and **Initargs** tabs to show how the instances can be accessed and initialized.
- **Functions** tab to show the generic functions or methods defined on this class, either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

Figure 5.8 KnowledgeWorks Objects Browser



Current cycle number: 6 Show Inferencing State: :DEFAULT

Preset query/pattern: NODE

Query: (NODE ?object)

Pattern:

```
(NODE #<NODE 4220105B3B>)
(NODE #<NODE 4220105B73>)
(NODE #<NODE 4220105BAB>)
(NODE #<NODE 4220105BE3>)
(NODE #<NODE 4220105C1B>)
```

```
CLOS object: Existed at start
Slots:
  ANIMAL : NIL
  QUESTION : "Does it have stripes?"
  YES-NODE : #<NODE 4220105CC3>
  NO-NODE : #<NODE 4220105D33>
```

Ready.

The Objects Browser is obtained by choosing **Window > KnowledgeWorks > Objects**. Any **<expression>** (See Section 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**.

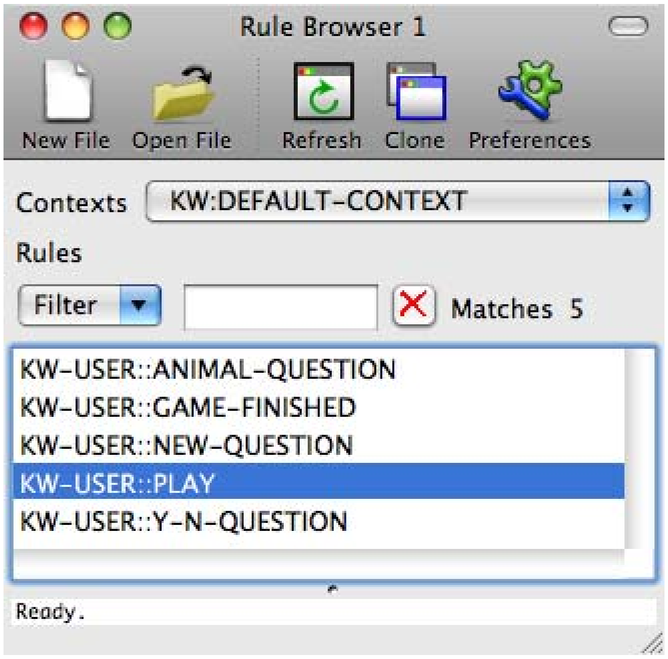
The package used to read and print symbols may be modified by choosing **LispWorks > Preferences... > Objects Browser > Package** and entering a package name into the Package pane. Clicking **OK** will update the tool.

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 **Window > Refresh**.

5.7 The Rule Browser

Figure 5.9 KnowledgeWorks Rule Browser



KW:DEFAULT-CONTEXT

KW-USER::ANIMAL-QUESTION
KW-USER::GAME-FINISHED
KW-USER::NEW-QUESTION
KW-USER::PLAY
KW-USER::Y-N-QUESTION

The Rule Browser may be obtained by choosing **Window > 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 Section 5.8, “Debugging with the Environment”) for the context, displaying the meta-interpreter (see Section 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 Section 5.8, “Debugging with the Environment”).

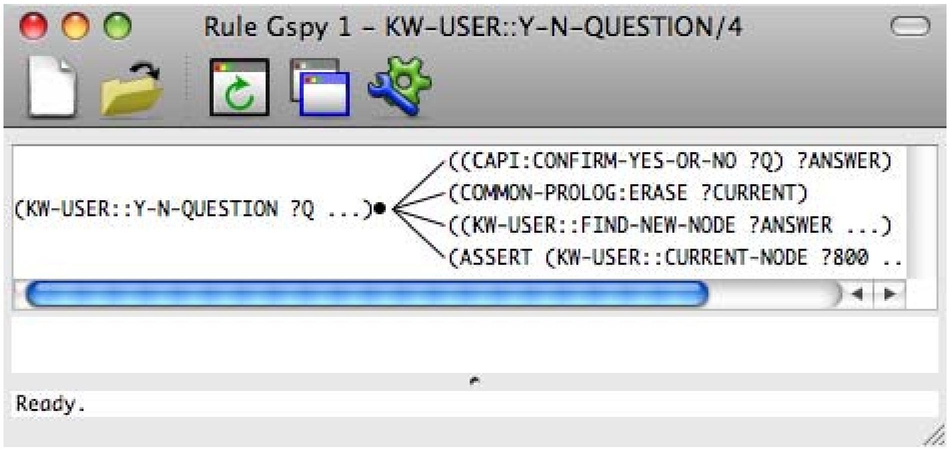
The **Rule** menu acts on the rule selected in the lower pane. All rules may be edited by choosing **Rule > Find Source**. Spy Windows can be brought up or removed by choosing **Rule > Gspy**. Forward chaining rules may have Monitor Windows (see Section 5.8 on page 62) brought up or removed by choosing **Rule > Monitor** (this command is disabled when a backward chaining rule has been selected). These are explained in Section 5.8, “Debugging with the Environment”.

The package used for displaying symbols may be modified by choosing **LispWorks > 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

Figure 5.10 KnowledgeWorks Gspy Window


Rule Gspy 1 – KW-USER::Y-N-QUESTION/4
(KW-USER::Y-N-QUESTION ?Q ...)
((CAPI:CONFIRM-YES-OR-NO ?Q) ?ANSWER)
(COMMON-PROLOG:ERASE ?CURRENT)
((KW-USER::FIND-NEW-NODE ?ANSWER ...)
(ASSERT (KW-USER::CURRENT-NODE ?800 ..
Ready.

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.

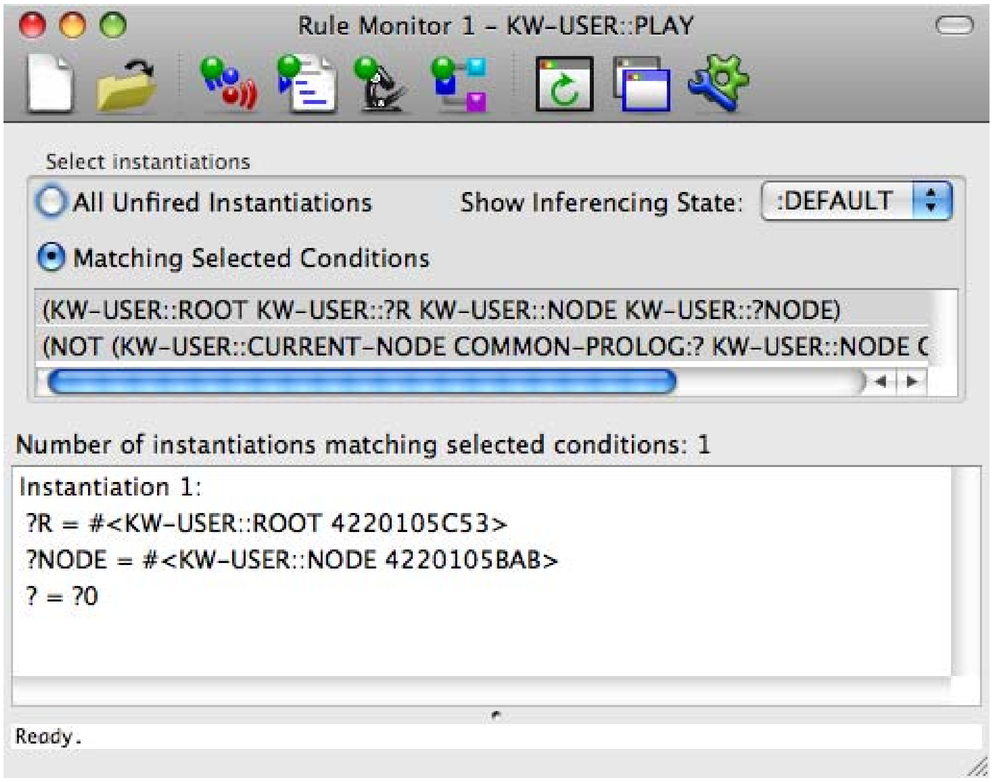
Selecting one of the graph nodes in the top pane of the Spy Window displays the full text of the box in the pane below. Choosing **Gspy** from the context menu brings up a Spy Window for the goal in the box.

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

Figure 5.11 KnowledgeWorks Rule Monitor



All Unfired Instantiations [✓ Matching Selected Conditions]

Show Inferencing State: :DEFAULT

```
(KW-USER::ROOT KW-USER::?R KW-USER::NODE KW-USER::?NODE)
(NOT (KW-USER::CURRENT-NODE COMMON-PROLOG:? KW-USER::NODE C
```

Number of instantiations matching selected conditions: 1

```
Instantiation 1:
 ?R = #<KW-USER::ROOT 4220105C53>
 ?NODE = #<KW-USER::NODE 4220105BAB>
 ? = ?0
```

Ready.

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, say,

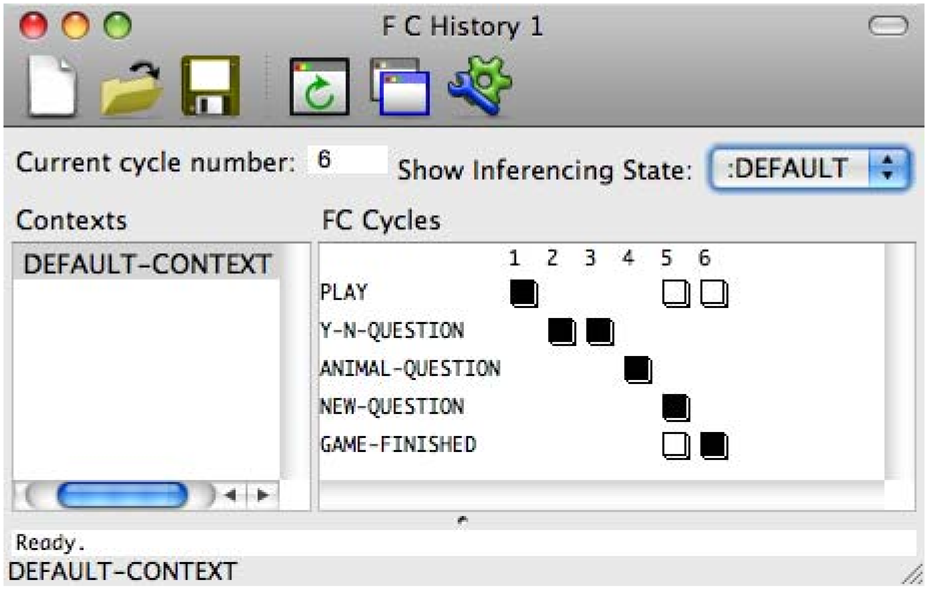
```
(person ?person1 father ?person)
(person ?person2 son ?person)
(test (not (eq ?person nil)))
```

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

Figure 5.12 KnowledgeWorks Forward Chaining History



Current cycle number: 6 Show Inferencing State: :DEFAULT

DEFAULT-CONTEXT

	1	2	3	4	5	6
PLAY	■				□	□
Y-N-QUESTION		■	■			
ANIMAL-QUESTION				■		
NEW-QUESTION					■	
GAME-FINISHED					□	■

Ready.
DEFAULT-CONTEXT

The Forward Chaining History may be viewed by choosing **Window > Knowl-edgeWorks > 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 Section 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:

```
(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

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

This implicitly defines the rule

```
(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

```
(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
            :title "Select an Instantiation:"
            :name-function #'(lambda (inst)
                                (format nil "~S: ~S"
                                  (inst-rulename inst)
                                  (inst-bindings inst)))))
```

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 Appendix 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"
                   ?l ?d ?t ?c ?loc)
             :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
         (do-the-rest explain-info
            (inst-bindings 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 Appendix 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 Section 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)))
```

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

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

  A rule could be defined similarly to:

```
(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:

```
(* (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

```
(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

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

and

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

both return **`t`**. Consequently, for any single given instantiation

```
(<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
- 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 colour)
(make-instance 'train :position 0 :speed 80)
(make-instance 'signal :position 10 :colour '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 5)))
```

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.

6.2.2 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`**, and
- *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:

```
(defrule age-of :backward
                ((age-of charlie 30) <--)
                ((age-of william 25) <--)
                ((age-of james 28) <--))
```

The call: `(age-of james ?x)` would jump directly to the third clause and bind `?x` to 28 without trying the other two.

The call: `(age-of tom ?x)` would fail immediately without doing any pattern matching.

Clauses are distinguished first by the types and then the values of their first arguments.

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:

```
(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:

```
(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:

```
(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 maintainence*.

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`**, page 100).

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

```
(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:

```
(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)))
         (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.
Signature	`all-debug`
Description	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.

Signature `any` ***pattern-to-instantiate goal-to-prove*** `=>` ***result*****,** ***successp***

Arguments

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

Values

result	`nil` or a value matching *pattern-to-instantiate*
successp	A boolean.

Description The backward chaining inference engine is started to look for any set of bindings which satisfy *goal-to-prove*. Using those bindings, *pattern-to-instantiate* is instantiated and returned.

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

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

Examples

```
(any '(?x is in (1 2 3)) '(member ?x (1 2 3)))
```

returns `(1 IS IN (1 2 3)), T`

```
(any '(?truck is a truck) '(truck ?truck))
```

returns `(#<TRUCK TRUCK5> IS A TRUCK), T`

See also `findall`

assert *Backward Chaining Goal*

Summary Creates or modifies objects in the object base.

Signature **`assert (`*class-name variable* {*slot-name term*}`*)`**

Arguments

class-name	The name of a class.
variable	A variable beginning with **`?`**.
slot-name	The name of a slot in the *class-name*.
term	An expression.

Description The *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 the *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.

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.

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.

Signature **`conflict-set`** ***variable***

Arguments	*variable*	An unbound KnowledgeWorks variable introduced by **`?`**.

Description

This backward chaining goal is only relevant when writing a meta-interpreter for a context. It binds *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

```
(conflict-set ?conflict-set)
```

See also

`instantiation`
`fire-rule`

context *Backward Chaining Goal*

Summary

Adds new contexts to the agenda.

Signature

`context` ***context-list***

Arguments	*context-list*	A list of context names.

Description

The given list of contexts in *context-list* is placed 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 *context-list* contains variables, then they must be already bound.

Examples

```
(context (my-context))
(context (?x ?y)) ; if ?x ?y bound to context names
```

See also

`return`

current-cycle *Function*

Summary
Returns the current forward chaining cycle number.

Signature
`current-cycle => `*cycle-number*

Values
cycle-number An integer.

Description
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.

Signature
`cut`

Description
`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
```
(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.

Description Deprecated. New code should use **`current-cycle`**.

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

See also **`current-cycle`**

def-kb-class *Macro*

Summary Defines a class for use in the object base.

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

Arguments The arguments are identical to those for **`defclass`**.

Values *class* The named class object.

Description Defines a new CLOS class as **`defclass`** does. However, if none of the given superclasses is a subclass of **`standard-kb-object`**, then **`standard-kb-object`** is added to the list of superclasses.

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.

Signature
: **`def-kb-struct`** ***name-and-options*** {***slot-description***}**`*`** **`=>`** ***name***

Arguments
: The arguments are as for **`defstruct`**, except that in *name-and-options* the only valid options are **`:include`** and **`:print-function`**.

Values

name	The name of the structure class.

Description
: Defines a KnowledgeWorks structure class. Objects of these classes are analogous to Lisp structures except that they may be used in rules similarly to CLOS objects.

Examples

```
(def-kb-struct start)
(def-kb-struct (named-kb-struct
  (:print-function print-named-kb-struct))
  (name (gensym 'named-kb-struct)))

(def-kb-struct (possible-trucks-for-load
  (:include named-kb-struct))
  load trucks)
```

See also

`def-kb-class`

def-named-kb-class *Macro*

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

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

Arguments The arguments are identical to those for **`defclass`**.

Values *class*	the named class object

Description Defines a new CLOS class as **`defclass`** does. However, if none of the given superclasses is a subclass of **`named-kb-object`**, then **`named-kb-object`** is added to the list of superclasses. The class inherits a name slot **`kb-name`** of which the initialization form (**`:initform`**) generates a symbol from the class name using **`(gentemp`** ***class-name*****`)`**.

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`

defcontext *Macro*

Summary Defines a context.

Signature **`defcontext`** ***context-name*** **`&key (`*****refractoriness*** **`t) (`*****auto-return*** **`t)`** ***strategy meta documentation***

Arguments	
context-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

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.

The *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.

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

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

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

Examples

```
(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`

```
-priority
priority
-recency
recency
-specificity
specificity
```

defrule *Macro*

Summary Defines a rule.

Signature **`defrule`** ***rule-name direction [doc-string]*** **`&rest`** ***body*** **`=>`** ***rule-name***

Arguments

rule-name	A symbol.
direction	Either **`:forward`** or **`:backward`**.
doc-string	An optional string.
body	Forms as described in Chapter 3, Rules.

Values

rule-name	A symbol.

Description Defines a rule named *rule-name* (which must be distinct from any other rule name, context name or KnowledgeWorks class name). If *direction* is **`:forward`** a forward chaining rule is defined, if **`:backward`** a backward chaining rule is defined. 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`**.

A full description is given in Chapter 3, Rules.

Examples

```
(defrule move-train :forward :context trains
  (train ?train position ?train-pos)
  (signal ?signal position ?signal-pos colour green)
  (test (= ?signal-pos (1+ ?train-pos)))
  -->
  ((format t "~%Train moving to ~S" ?signal-pos))
  (assert (signal ?signal colour 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)))
```

deftactic *Macro*

Summary Defines a tactic function for use in context strategies.

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 Defines a new conflict resolution tactic named *tactic-name*. The *type* of the tactic may be `:static` if the body does not look into the slots of the objects making up the instantiation, otherwise `:dynamic`. The *lambda-list* binds to two instantiation objects and the function body *body* should return non-nil if and only if the first instantiation object is preferred to the second. **deftactic** also defines a function of the same and *body* can be preceded by a documentation string.

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

Examples

```
(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)))))
```

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.

Signature

destroy-inferencing-state ***name-or-state***

Arguments

name-or-state Any object.

Description

Destroys an inferencing state named by *name-or-state.*

If *name-or-state* is and 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 *Backward Chaining Goal*

Summary Erases an object from the object base.

Signature **`erase`** ***variable***

Arguments *variable* A a KnowledgeWorks object.

Description The *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 *Backward Chaining Goal*

Summary The standard prolog predicate that always fails.

Signature **`fail`**

Description This goal 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`

find-inferencing-state *Function*

Summary Finds a known inferencing state.

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 Finds and returns an inferencing state named by *name*.

If an inferencing state with the same name already exists (as compared using **`eql`**), it is returned.

Otherwise, the value of *if-does-not-exist* determines what happens:

`:error`	A continuable error is signaled. Invoking the **`continue`** restart creates and returns a new inferencing state.
`:create`	A new inferencing state is created and returned.

Examples

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

See also

`destroy-inferencing-state`
`*inferencing-state*`
`inferencing-state-name`
`list-all-inferencing-states`
`make-inferencing-state`

findall *Function*

Summary

Return all matches of a backward chaining goal.

Signature

`findall` ***pattern-to-instantiate goal-to-prove*** **`=>`** ***list***

Arguments

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

Values

list	A list

Description

The backward chaining inference engine is started 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.

Examples

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

returns

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

```
(findall '(?truck is a truck) '(truck ?truck))
```

returns

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

See also **`any`**

fire-rule *Backward Chaining Goal*

Summary .Fires the given meta-interpreter rule instantiation.

Signature `fire-rule` ***instantiation***

Arguments *instantiation* An instantiation object.

Description This backward chaining goal is only relevant when writing a meta-interpreter for a context. It fires the given rule instantiation. It is an error if the passed object is not an instantiation object.

Examples

```
(fire-rule ?instantiation)
```

See also

`start-cycle`
`instantiation`
`defcontext`
`standard-context`

get-kb-object *Function*

Summary Finds a named object in the object base.

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

Arguments *object-name* A symbol.

Values *object* A KnowledgeWorks CLOS object.

Description	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`**

in-interpreter *Variable*

Summary	Allows code to detect when it is running in a rule.
Description	The variable is bound to **`t`** if the code executing has been called (directly or indirectly) from the forward chaining rule interpreter. Otherwise it bound to **`nil`**. The value should not be changed.
Initial Value	**`nil`**

infer *Function*

Summary	Runs the forward chaining inferencing engine.	
Signature	**`infer &key (`*`contexts`*` '(default-context)) => `*`cycle-count`***	
Arguments	*contexts*	A list of context names.
Values	*cycle-count*	An integer.

Description

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.

Description

The value of **`*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.

Initial Value

An empty inferencing state named **`:default`**.

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.

Signature	**`inferencing-state-name`** ***state*** **`=>`** ***name***	
Arguments	*state*	An inferencing state.
Values	*name*	Any object.
Description	Returns the name of the given inferencing state.	
Examples	**`(inferencing-state-name *inferencing-state*)`**	
See also	**`find-inferencing-state`** **`*inferencing-state*`** **`list-all-inferencing-states`** **`make-inferencing-state`**	

inst-bindings *Function*

Summary	Returns the bindings in a rule instantiation.	
Signature	**`inst-bindings`** ***instantiation*** **`=>`** ***bindings***	
Arguments	*instantiation*	An instantiation object.
Values	*bindings*	An association list.
Description	Returns an association list of the variables and their bindings in the *instantiation*. The variables are those produced by the condition part of the forward chaining rule.	

Example

For an instantiation of a rule with the precondition

```
(object ? color ?color-value size ?size)
```

the value returned by

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

might be

```
((?color-value . :red) (?size . 20))
```

See also
`conflict-set`
`deftactic`
`inst-rulename`
`inst-token`
`instantiation`

inst-rulename *Function*

Summary
Returns the rule name of a rule instantiation.

Signature
`inst-rulename` ***instantiation*** **`=>`** ***rulename***

Arguments

instantiation	An instantiation object.

Values

rulename	A symbol which is the name of a rule.

Description
Returns the rule name of the *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.

Signature
`inst-token` ***instantiation*** **`=>`** ***token***

Arguments

instantiation	An instantiation object.

Values	*token*	A list of objects.

Description

Returns the token of the *instantiation*. The *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. For example, if the forward chaining conditions are

```
(train ?train)
(signal ?signal)
```

then the token will have the form `(`*signal-object train-object*`)`.

See also

`conflict-set`
`deftactic`
`inst-rulename`
`inst-bindings`
`instantiation`

instantiation *Backward Chaining Goal*

Summary

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

Signature

`instantiation` ***variable***

Arguments	*variable*	An unbound variable introduced by **`?`**.

Description

This backward chaining goal is only relevant when writing a meta-interpreter for a context. It binds *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

```
(instantiation ?instantiation)
```

See also

`conflict-set`
`inst-bindings`
`inst-rulename`
`inst-token`
`start-cycle`
`fire-rule`
`defcontext`
`standard-context`

list-all-inferencing-states *Function*

Summary

Returns a list of all the known inferencing states.

Signature

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

Values

states	A list of inferencing states.

Description

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

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

See also

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

kb-name *Generic Function*

Summary

Returns the name of an object.

Signature

`kb-name` ***object*** **`=>`** ***name***

Arguments	*object*	A KnowledgeWorks named CLOS object
Values	*name*	A symbol.

Description

Returns the name of *object*. It is an error if the object is not a named object. Classes of named objects can be defined using the macro **`def-named-kb-class`**.

Examples

```
(kb-name (get-kb-object 'fred))  ; returns FRED
```

See also

`def-named-kb-class`
`get-kb-object`
`named-kb-object`

kw-class *Backward Chaining Goal*

Summary

Matches all KnowledgeWorks class names.

Signature

`kw-class` ***term***

Arguments	*term*	Any backward chaining term.

Description

This goal can act as a generator and is resatisfiable. It succeeds when *term* is a symbol which is the name of a KnowledgeWorks class. If *term* is an unbound variable it generates the names of the KnowledgeWorks classes.

Examples

```
(kw-class truck)  ; succeeds if truck is a KW class

(kw-class ?class)
  ; ?class is bound to the name of a KW class
```

See also

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

-lex *Conflict Resolution Tactic / Function*

Summary

Implements the **`-lex`** tactic.

Signature

`-lex` ***instantiation1 instantiation2*** `=>` ***result***

Arguments

instantiation1	An instantiation object.
instantiation2	An instantiation object.

Values

result	A boolean.

Description

The function 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

`defcontext`
`deftactic`
`lex`
`instantiation`
`conflict-set`
`fire-rule`

lex *Conflict Resolution Tactic / Function*

Summary

Implements the **`lex`** tactic.

Signature

`lex` ***instantiation1 instantiation2*** `=>` ***result***

Arguments

instantiation1	An instantiation object.
instantiation2	An instantiation object.

Values

result	A boolean.

Description

The function 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

`defcontext`
`deftactic`
`-lex`
`instantiation`
`conflict-set`
`fire-rule`

make-inferencing-state *Function*

Summary

Makes a new inferencing state.

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

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 *Generic Function*

Summary

Makes a CLOS or KnowledgeWorks structure object.

Signature

`make-instance` ***class*** **`&rest`** ***initargs*** **`=>`** ***object***

Arguments

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

Arguments

object	A new instance of *class*.

Description

A new instance of the class *class* is made. The class may be either a CLOS class or a KnowledgeWorks structure class, in which case the *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.

Signature

-mea ***instantiation1 instantiation2*** **=>** ***result***

Arguments

instantiation1	An instantiation object.
instantiation2	An instantiation object.

Values

result	A boolean.

Description

The function returns true if and only if *instantiation1* is preferred to *instantiation2* by the conflict resolution tactic **-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

defcontext
deftactic
mea
instantiation
conflict-set
fire-rule

mea

Conflict Resolution Tactic / Function

Summary

Implements the **mea** tactic.

Signature

mea ***instantiation1 instantiation2*** **=>** ***result***

Arguments	*instantiation1*	An instantiation object.
	instantiation2	An instantiation object.
Values	*result*	A boolean.

Description The function returns true if and only if *instantiation1* is preferred to *instantiation2* by the conflict resolution tactic **`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 **`defcontext`**
`deftactic`
`-mea`
`instantiation`
`conflict-set`
`fire-rule`

named-kb-object *Class*

Summary A class that provides named objects.

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 This class is the mixin class for named KnowledgeWorks CLOS objects.

Subclasses of **`named-kb-class`** 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.

Signature

no-debug

Description

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.

Signature

not {*condition*}*

Arguments

condition	Any backward chaining goal.

Description

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

If **not** is used in a forward chaining pre-condition, it succeeds if any of the *condition* contained within it fail. In this usage, the *condition*s may only contain expressions normally allowed in forward chaining pre-conditions (object base references and lisp tests).

Examples

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

See also

test

-order

Conflict Resolution Tactic / Function

Summary

Implements the **-order** tactic.

Signature

-order ***instantiation1 instantiation2*** **=>** ***result***

Arguments

instantiation1	An instantiation object.
instantiation2	An instantiation object.

Values

result	A boolean.

Description

The function 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

`defcontext`
`deftactic`
`order`
`instantiation`
`conflict-set`
`fire-rule`

order — *Conflict Resolution Tactic / Function*

Summary

Implements the **`order`** tactic.

Signature

`order` ***instantiation1 instantiation2*** **`=>`** ***result***

Arguments

instantiation1	An instantiation object.
instantiation2	An instantiation object.

Values

result	A boolean.

Description

The function 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

`defcontext`
`deftactic`
`-order`
`instantiation`
`conflict-set`
`fire-rule`

print-verbose *Variable*

Summary
: Controls how much information is printed for an object.

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

Initial Value
: **`nil`**

-priority *Conflict Resolution Tactic / Function*

Summary
: Implements the **`-priority`** tactic.

Signature
: **`-priority`** ***instantiation1 instantiation2*** **`=>`** ***result***

Arguments

instantiation1	An instantiation object.
instantiation2	An instantiation object.

Values

result	A boolean.

Description
: The function 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

`defcontext`
`deftactic`
`priority`

instantiation
conflict-set
fire-rule

priority *Conflict Resolution Tactic / Function*

Summary

Implements the **`priority`** tactic.

Signature

`priority` ***instantiation1 instantiation2*** **`=>`** ***result***

Arguments

instantiation1	An instantiation object.
instantiation2	An instantiation object.

Values

result	A boolean.

Description

The function returns true if and only if *instantiation1* is preferred to *instantiation2* by the conflict resolution tactic **`priority`**, otherwise **`nil`**. 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

defcontext
deftactic
-priority
instantiation
conflict-set
fire-rule

-recency *Conflict Resolution Tactic / Function*

Summary

Implements the **`-recency`** tactic.

Signature	`recency` ***instantiation1 instantiation2*** `=>` ***result***	
Arguments	*instantiation1*	An instantiation object.
	instantiation2	An instantiation object.
Values	*result*	A boolean.

Description

The function 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

`defcontext`
`deftactic`
`recency`
`instantiation`
`conflict-set`
`fire-rule`

recency *Conflict Resolution Tactic / Function*

Summary

Implements the **`recency`** tactic.

Signature	`recency` ***instantiation1 instantiation2*** `=>` ***result***	
Arguments	*instantiation1*	An instantiation object.
	instantiation2	An instantiation object.
Values	*result*	A boolean.

Description

The function returns truel 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

defcontext
deftactic
-recency
instantiation
conflict-set
fire-rule

reset *Function*

Summary

Clears all objects from the object base.

Signature

reset

Description

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.
Signature	`return`
Description	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.

Description

Determines behavior when creating a new named KB object with the same name as an existing KB object.

The possible values are:

`:error`	Signals a 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.

Initial Value

`:error`.

-specificity *Conflict Resolution Tactic / Function*

Summary Implements the **`-specificity`** tactic.

Signature **`-specificity`** ***instantiation1 instantiation2*** **`=>`** ***result***

Arguments

instantiation1	An instantiation object.
instantiation2	An instantiation object.

Values

result	A boolean.

Description The function returns truel 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

`defcontext`
`deftactic`
`specificity`
`instantiation`
`conflict-set`
`fire-rule`

specificity *Conflict Resolution Tactic / Function*

Summary Implements the **`specificity`** tactic.

Signature **`specificity`** ***instantiation1 instantiation2*** **`=>`** ***result***

Arguments

instantiation1	An instantiation object.
instantiation2	An instantiation object.

Values	*result*	A boolean.

Description

The function 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

`defcontext`
`deftactic`
`-specificity`
`instantiation`
`conflict-set`
`fire-rule`

standard-context *Backward Chaining Goal*

Summary

The standard meta-interpreter context.

Signature

`standard-context`

Description

A built-in backward chaining goal which implements a meta-interpreter for the default (normal) behavior of a context. It is as if defined by the rule

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

Examples

```
(defcontext 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.

Superclasses **standard-object**

Description This class is the mixin class for (unnamed) KnowledgeWorks CLOS objects.

Subclasses of **standard-kb-class** 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.

Signature **start-cycle**

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

Example
`(start-cycle)`

See also
`fire-rule`
`instantiation`
`defcontext`
`standard-context`

start-kw *Function*

Summary
Starts the KnowledgeWorks programming environment.

Signature
`start-kw`

Description
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.

Example
`(start-kw)`

test *Backward Chaining Goal*

Signature
`test` ***lisp-form***

Arguments

lisp-form	A single lisp form.

Description
Succeeds if and only if the *lisp-form* returns a non-nil value. Any currently bound variables may be used in the lisp form.

Examples

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

undefcontext *Macro*

Summary
Removes a named context and its rules.

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

Arguments

context-name	A symbol which names a context.
ignore	Ignored arguments.

Description
Removes the context named *context-name* and all the rules in it.

The *ignore* arguments are provided so that "un" may be prepended to a context definition in an editor buffer and evaluated to remove the context.

Examples

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

See also
`defcontext`

undefrule *Macro*

Summary
Removes a rule.

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

Arguments

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

Description
Removes the rule named *rule-name* and any unfired instantiations of that rule.

The *ignore* arguments are provided so that "un" may be prepended to a rule definition in an editor buffer and evaluated to remove the rule.

Examples

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

See also **`defrule`**

with-rule-actions *Macro*

Summary Allows rule syntax to be embedded in Lisp code.

Signature **`with-rule-actions`** ***bound-variables*** **`&body`** ***body*** **`=>`** ***successp***

Arguments

bound-variables	A list of variables (each starting with **`?`**)
body	A rule body.

Values

successp	A boolean.

Description This macro enables rule syntax to be embedded within Lisp. The *body* is executed just as if it were the right hand side of a forward or backward chaining rule. All variables in the body (each starting with **`?`**) are taken to be unbound unless found in the list *bound-variables*, in which case its value is taken from the Lisp variable of the same name. It is similar to the function **`any`** but can be compiled for efficiency.

The value *successp* is **`t`** if the body succeeds (that is, all clauses are successfully executed) or **`nil`** if any of the clauses fail.

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

Example

```
(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)))
```

See also **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 *LispWorks 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:

```
(defrel append
  ((append () ?x ?x))
  ((append (?u . ?x) ?y (?u . ?z))
   (append ?x ?y ?z)))
```

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))

?X = NIL
?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:

```
?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 ...>
|
|YES.
|OK.
|
|==> (color ?x)
|
|?X = RED
```

A.4.4 Exiting the Interpreter

The Common Prolog interpreter may be exited by typing:

```
|==> (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

```
|==> ((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>
```

```
|==> ((typep 3 'integer) ?x)
|
|?X = T
|
|==> ((typep 3 'integer) t)
|
|YES.
|
|==> (and ((floor 5 3) ?x) ((floor 4 3) ?x))
|
|?X = 1
|
|==> ((cons 3 4) (?x . ?y))
|
|?X = 3
|?Y = 4
|
|==> (and (= ?op *) ((list ?op 3 4) ?y) (call (?y ?z)))
|
|?OP = *
|?Y = (* 3 4)
|?Z = 12
|
|==> (and (defrel fact
|            ((fact 0 1))
|            ((fact ?x ?y)
|             ((- ?x 1) ?w)
|             (fact ?w ?z)
|             ((* ?z ?x) ?y)))
|          (fact 10 ?result))
|
|?X = ?0
|?Y = ?1
|?W = ?2
|?Z = ?3
|?RESULT = 3628800
```

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:

```
(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:

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

returns:

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

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

returns:

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

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

returns:

```
((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

`(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 mention in *args* appears in *sample-expr* or *return-expr*, then its value is subsituted. All other symbols in *sample-expr* and *return-expr* remain unchanged.

A.6.2.3 with-prolog

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

```
(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:

```
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 “A Simple Explanation Facility” on page 72.

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 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, 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)bort     - 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:
|
|(DEFREL MEMBER
|  ((MEMBER ?X (?X . ?)))
|  ((MEMBER ?X (? . ?Y)) (MEMBER ?X ?Y))) ? c
|  ...user selects creep

|[1] EXIT: (MEMBER 3 (3 . ?0))? r
|  ...user selects retry
|
|[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 . ?0))? l <- leap

|?X = (3 . ?0); <- more solutions
|
|[1] REDO: (MEMBER 3 (3 . ?0))? 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
|?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 NIL ?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:

```
(defrelmacro <name> <arg-list> <body>)
```

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

A.8.1 Example

```
(defrelmacro append3 (x y z w)
  (let ((iv (make-internal-var)))
  `(and (append ,x ,y ,iv)
        (append ,iv ,z ,w))))

==> (append3 (1) (2) (3) ?y)

?Y = (1 2 3)
```

A.9 Defining Definite Clause Grammars

The **`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 *sentence*) matches a particular grammar. The remaining tokens in the list become the *sentence tail*.

The relation has the form

```
(<grammar name> <sentence> <sentence tail> <extra argument>*)
```

where the **`<extra argument>`** items are terms defined below.

The syntax of the defgrammar macro is

```
(defgrammar <grammar name>
  <rule>*)

<rule> ::= (<lhs> <rhs>*)
```

```
<lhs> ::= <grammar name>
        | (<grammar name> <term>*)

<rhs> ::= <atom>
        | <var>
        | (<other grammar name> <term>*)
        | <lisp clause>
        | (call <term>)
        | (cut)

<lisp clause> ::= (<non-atomic lisp form> <term>*)

<non-atomic lisp form> ::= (<lisp function name> <lisp arg>*)
```

`<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>` is 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.

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.

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)))

(defgrammar gram-vp
  (gram-vp (gram-v) (gram-np))
  (gram-vp (gram-v)))

(defgrammar gram-s
  (gram-s (gram-np) (gram-vp)))
```

Note the use of symbols for terminals and lists for non-terminals. They all use the first form of the **`<lhs>`** and have no extra terms on the **`<rhs>`**, so all of the relations are binary.

The following will both succeed and bind **`?x`** to the list **`(foo bar)`**:

```
(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.9.1.2 Example 2: Using extra arguments.

```
(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)`**:

```
(clog:any '?x '(two-of (start start foo bar) ?x start))
```

The following will both fail because the sentences do not begin with two **`start`** symbols:

```
(clog:any '?x '(two-of (not-start start foo bar) ?x start))
(clog:any '?x '(two-of (start not-start foo bar) ?x start))
```

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 reconsult 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 reconsult 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 for users with bitmap displays. The environment consists 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

`/== (?x ?y)`	same as Prolog `\==`
`= (?x ?y)`	standard Prolog
`=.. (?x ?y)`	standard Prolog
`== (?x ?y)`	standard Prolog
`@< (?x ?y)`	same as Prolog except all variables sort as identical
`@=< (?x ?y)`	ditto
`@> (?x ?y`	ditto
`@>= (?x ?y)`	ditto
`append (?x ?y ?z)`	standard Prolog
`arg (+index +term ?value)`	standard Prolog
`asserta (+exp)`	standard Prolog
`assertz (+exp)`	standard Prolog
`atomic (?x)`	standard Prolog
`bagof (?exp (+goal . +ex-vars) ?bag)`	standard Prolog (unusual syntax)*
`call (+exp)`	standard Prolog
`clause (+head ?tail)`	standard Prolog
`debug ()`	cause debugging information to be saved for each call whether it is spied or not
`debugging ()`	display a list of all spied goals
`defdetrel (+name &rest +clauses)`	define a relation and declare it to be deterministic
`defgrammar (+name &rest +rules)`	define a grammar rule
`defrel (+name &rest +clauses)`	define a relation
`defrelmacro (+name +args &rest +body)`	define a logic macro

`defrel-special-form-macro (+name +args &rest +body)`	like defrelmacro but can have &rest in **`+args`**. Use of this form will shadow all predicates named **`+name`** regardless of arity.
`deterministic (+name)`	declare the relation called **`?name`** to be deterministic
`erase (+ref)`	delete the predicate with database reference **`?ref`** from the database
`fail ()`	standard Prolog
`findall (?exp +goal ?result)`	generate all solutions to ?goal and instantiate **`?exp`** with the values. Return a list in **`?result`**.
`findallset (?exp +goal ?result)`	same as findall/3 but removes duplicates
`functor (?term ?functor ?arity)`	standard Prolog
`halt ()`	exit Common Prolog
`integer (?x)`	standard Prolog
`is (?result +exp)`	standard Prolog
`keysort (+in ?out)`	standard Prolog except uses alist style cons pairs
`leash (+event-spec)`	cause the interpreter to pause and ask for input when one of the leashed events is traced. An event-spec is one of: **`(call`** exit redo fail), or a list of ports.
`listing (+name &optional +arity)`	display a listing of the named predicate or listings for each arity if no arity is specified
`member (?x ?y)`	standard Prolog
`nodebug ()`	leave debug mode (cease saving debug info for non-spied goals)
`nonvar (?x)`	standard Prolog
`nospy (+args)`	remove **`+args`** from the list of spied goals. +args may be a predicate name or a list of predicate names. Unspy all goals if **`+args`** is nil
`not (+x)`	standard Prolog

`notrace ()`	turn off exhaustive tracing for debugged goals
`once (+exp)`	satisfy **`+exp`** as a goal once, then fail on retrying even if **`+exp`** has more solutions: this can be used to make a call deterministic so that the compiler can perform last call optimization
`output-defrels (+name ?defrels)`	return a list of **`defrel`** expressions derived from the dynamic clauses associated with **`?name`**
`read-term (?term)`	read in a term
`recorda (+exp ?val ?ref)`	standard Prolog
`recorded (+term ?val ?ref)`	standard Prolog
`recordz (+exp ?val ?ref)`	standard Prolog
`repeat ()`	standard Prolog
`retract (+clause)`	standard Prolog
`setof (?exp (+goal . +ex-vars) ?bag)`	standard Prolog (unusual syntax)*
`sort (+in ?out)`	standard Prolog
`spy (+args)`	spy **`+args`**. **`+args`** may be a predicate name or a list of predicate names. If arity is not mentioned for a predicate name, predicates of all aritys with that name are spied.
`trace ()`	turn on tracing for debugged goals, also turn on debugging for the next top level goal
`translate-vars (?intern ?extern)`	translate back and forth between internal and external variable representations. Can be used to pretty up the writing of terms containing variables

`true ()`	standard Prolog
`unleash (+event-spec)`	Undo leashing for **`+event-spec.`** +event-spec may be a port or a list of ports. If **`+event-spec`** is nil, all ports are unleashed.
`var (?x)`	standard Prolog

* **`setof`** and **`bagof`** in standard Prolog use a special syntax for existentially quantified variables, for example:

```
?- setof(X, Y^foo(X,Y), Z).
```

In Common Prolog, this would look like:

```
==> (setof ?x ((foo ?x ?y) ?y) ?z)
```

So, a goal with no existentially quantified variables is nested in an extra set of parentheses:

```
==> (bagof ?x ((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 doesn’t have to unify any variables. Arguments are referenced with: **`(special-arg`** <*argnum*>**`)`**. 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:

```
-->
->
/
//
<<
=
=
=<
>>
?-
@<
@>
@>=
\,
\.
\:-
\:=
\;
\\
\\
\\+
\\/
\\=
\\==
^
current-op
display
get
get0
is
name
nl
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 (Chapter 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))

;;; ask an animal question - these a leaf questions

(defrule animal-question :forward
  (current-node ?current node ?node)
  (node ?node animal ?animal question nil)
  -->
  ((tk:confirm-yes-or-no
     (format nil "Is it a ~a?" ?animal)) ?answer)
  (erase ?current)
  (assert (game-over ? node ?node animal ?animal
    answer ?answer)))

;;; add new nodes to the tree for the new animal and
;;; the question that distinguishes it
```

```
(defrule new-question :forward
  :priority 20
  (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 ~
           answer is yes for a ~a and no for a ~a"
           ?new-animal ?animal)) ?question)
  (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
  :priority 15
  (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
     (doesnt-exist-already ?new-animal)
     (and ((tk:send-a-message "Animal exists already"))
          (fail)))))

;;; check if a node already refers to this animal
```

```
(defrule doesnt-exist-already :backward
  ((doesnt-exist-already ?animal)
   <--
   (node ? animal ?animal)
   (cut)
   (fail))
  ((doesnt-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 Chapter 6, “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 runtime
(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 runtime 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)))
    (if (is-var expr) (list expr) nil)))

(defun make-explain-func (explain-stuff)
  "generates a function to generate explanation text at
   runtime"
  (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
  :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 ?o1 kb-name ?w)
  -->
  ((format t "~%Walk to ~s" ?p))
  (assert (monkey ?m at ?p))
  (assert (object ?o1 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 on object ?o)
  (object ?o1 kb-name ?o at ?p)
  (monkey ?m at ?p holds nil)
  -->
  ((format t "~%Climb onto ~s" ?o))
  (assert (monkey ?m on ?o))
  (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
  :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 Chapter 6, “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 doesn't 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 doesn't need the meta
;;; -interpreter as that's only necesssary 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)))
             ; next context onto agenda

(defrule diagnose1 :forward
  :context diagnose
  (symptom ?s type over-heat)
  (fault ?f type lack-of-water)
  -->
  ((implication-strength 0.9))
  ((add-evidence ?f ?s)))

(defrule diagnose2 :forward
  :context diagnose
  (symptom ?s type overheat)
  (fault ?f type unknown)
  -->
  ((implication-strength 0.1))
  ((add-evidence ?f ?s)))

(defrule diagnose3 :forward
  :context diagnose
  (symptom ?s type wont-start)
  (fault ?f type battery)
  -->
  ((implication-strength 0.9))
  ((add-evidence ?f ?s)))
```

```
(defrule diagnose4 :forward
  :context diagnose
  (symptom ?s type wont-start)
  (fault ?f type unknown)
  -->
  ((implication-strength 0.1))
  ((add-evidence ?f ?s)))

(defrule diagnose5 :forward
  :context diagnose
  (symptom ?s type power-loss)
  (fault ?f type lack-of-oil)
  -->
  ((implication-strength 0.9))
  ((add-evidence ?f ?s)))

(defrule diagnose6 :forward
  :context diagnose
  (symptom ?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
  (symptom ?s1 type ?t1)
  (symptom ?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
;;; print the suggested remedies out in order

(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, and

- 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 runtime 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: `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). Whilst 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

- *Common LISPcraft*, by Robert Wilensky (Norton). An introductory text on Lisp.
- *Common Lisp the Language, Second Edition*, by Guy. L. Steele Jr. (Digital Press). This is the complete reference book on Common Lisp.
- *Object-Oriented Programming in Common Lisp*, by Sonya E. Keene (Addison-Wesley). An introductory text on CLOS for programmers.
- *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 addtion to the *KnowledgeWorks and Prolog User Guide*, the LispWorks manual set includes the following manuals which might be helpful whilst 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.
- 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 *LispWorks Editor User Guide* describes the keyboard commands and programming interface to the the LispWorks IDE editor tool.

These books are all available in HTML, PDF and Postscript formats.

- The *LispWorks 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.

This book is available in PDF and Postscript 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, “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)
  -->
  ((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 Appendix 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 the user 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.

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 the user 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