

Neuron Data Elements Environment Intelligent Rules Element

Version 4.1

Language Reference

© Copyright 1986–1997, Neuron Data, Inc. All Rights Reserved.

This software and documentation is subject to and made available only pursuant to the terms of the Neuron Data License Agreement and may be used or copied only in accordance with the terms of that agreement. It is against the law to copy the software except as specifically allowed in the agreement. This document may not, in whole or in part, be copied, photocopied, reproduced, translated, or reduced to any electronic medium or machine-readable form without prior consent, in writing, from Neuron Data, Inc.

Use, duplication, or disclosure by the U.S. Government is subject to restrictions set forth in the Neuron Data License Agreement and in subparagraph (c)(1) of the Commercial Computer Software-Restricted Rights Clause at FAR 52.227-19; subparagraph (c)(1)(ii) of the Rights in Technical Data and Computer Software clause at DFARS 252.227-7013; subparagraph (d) of the Commercial Computer Software—Licensing clause at NASA FAR supplement 16-52.227-86; or their equivalent.

Information in this document is subject to change without notice and does not represent a commitment on the part of Neuron Data. THE SOFTWARE AND DOCUMENTATION ARE PROVIDED “AS IS” WITHOUT WARRANTY OF ANY KIND INCLUDING WITHOUT LIMITATION, ANY WARRANTY OF MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE. FURTHER, NEURON DATA DOES NOT WARRANT, GUARANTEE, OR MAKE ANY REPRESENTATIONS REGARDING THE USE, OR THE RESULTS OF THE USE, OF THE SOFTWARE OR WRITTEN MATERIAL IN TERMS OF CORRECTNESS, ACCURACY, RELIABILITY, OR OTHERWISE.

Open Interface Element™, Data Access Element™, Intelligent Rules Element™, and Web Element™ are trademarks of, and are developed and licensed by Neuron Data, Inc., Mountain View, California. NEXPERT OBJECT® and NEXPERT® are registered trademarks of, and are developed and licensed by, Neuron Data, Inc., Mountain View, California.

Other brand or product names are the trademarks or registered trademarks of their respective holders.

Contents

Preface

Purpose of this Manual	vii
Description	vii
Audience	vii
How to Use this Manual	viii
Organization	x
Related Manuals	x

1. Application Development Features

ABS Function	1
ACOS Function.....	2
Actions	2
Agenda.....	5
ASIN Function.....	6
AskQuestion Operator	7
Assign Operator	8
ATAN Function.....	10
AVERAGE Function	11
Backward Chaining	12
Backward Operator.....	13
BOOL2STR Function	14
Boolean Constants.....	15
Boolean Expressions	16
Boolean Formats.....	17
CEIL Function.....	19
CHARFIND Function.....	19
Classes.....	20
Comment Attribute.....	22
COMPARE Function	23
Comparison Operators.....	24
Conditions.....	26
Context Links.....	28
COS Function.....	29
COSH Function	30
CreateObject Operator.....	30
Data Types	32
Data Validation Attribute	33
Date Formats.....	35
DATE Function.....	38
DATE2FLOAT Function	39
DATE2STR Function.....	40
DAY Function.....	41
DeleteObject Operator.....	42
Dynamic Data Exchange	43
Dynamic Objects	47
Execute Operator.....	48
Execute Routines	50

EXP Function	53
Expressions	54
FLOAT2DATE Function	56
FLOAT2INT Function	56
FLOAT2STR Function	57
FLOAT2TIME Function	58
Floating Point Constants.....	59
Floating Point Formats.....	60
FLOOR Function	63
Format Attribute	64
Formats.....	64
Forward Chaining.....	66
HOUR Function.....	68
Hypotheses.....	69
Identifiers	70
If Change Method	71
Inference	73
Inference Priority Attribute	74
Inference Slot Attribute	75
Inference Strategy	76
Inheritability Strategy.....	79
Inheritance.....	81
Inheritance Priority Attribute.....	82
Inheritance Slot Attribute.....	83
Inheritance Strategy.....	84
InhMethod Operator	86
InhValueDown Operator	87
InhValueUp Operator.....	88
Init Value Attribute.....	89
INT2STR Function	90
Integer Constants	91
Integer Formats	91
Interpretations	94
Interrupt Operator	95
LENGTH Function.....	96
LN Function	97
LoadKB Operator	98
LOG Function	100
MAX Function	101
Member Operator	102
Meta-Slots.....	103
Methods.....	104
MIN Function	107
MINUTE Function	109
MOD Function.....	109
MONTH Function.....	110
Multi-Values	111
No Operator	112
NoInherit Operator	113
NotMember Operator.....	113
NOW Function	114
Objects.....	115
Order of Sources Method.....	117
Patterns.....	120

POW Function	123
PROD Function	124
Prompt Line Attribute	125
Properties	126
Question Window Attribute.....	127
RAND Function.....	128
RANDOM Function.....	129
RANDOMMAX Function	130
RANDOMSEED Function.....	130
Reserved Words	131
Reset Operator.....	132
Retrieve Operator.....	133
ROUND Function	135
Rules.....	136
RunTimeValue Operator.....	137
SECOND Function.....	138
SELF	139
Semantic Gates	140
SendMessage Operator	141
Show Operator	145
SIGN Function	148
SIN Function	148
SINH Function.....	149
Slots	150
SQRT Function	152
STDEV Function.....	153
Strategy	154
Strategy Operator.....	155
STRCAT Function	157
STRFIND Function.....	158
String Constants	159
String Formats	160
STRLEN Function	162
STRLOWER Function.....	162
STRUPPER Function.....	163
STR2BOOL Function	164
STR2DATE Function.....	165
STR2FLOAT Function	166
STR2INT Function	167
STR2TIME Function.....	168
SUBSTRING Function	169
SUM Function.....	170
TAN Function	171
TANH Function.....	172
Time Formats	173
TIME Function.....	174
TIME2FLOAT Function	175
TIME2STR Function.....	176
UnloadKB Operator	177
Value Property.....	179
VAR Function	180
WEEKDAY Function	181
Why Attribute.....	182
Write Operator	183

YEAR Function	184
YEARDAY Function	185
Yes Operator	186
2. Execute Library Routines	
Execute Library Overview	187
Using The Execute Library	190
AtomExist Routine.....	197
AtomNameValue Routine	198
ComputeMultiValue Routine.....	200
ControlSession Routine.....	202
CopyFrame Routine.....	204
CreateObjects Routine.....	205
CreateReport Routine.....	207
Formatting Commands	209
Conditional Statements	211
Include Command	212
FileExist Routine	213
FindListElem Routine.....	214
GetListElem Routine.....	216
GetMultiValue Routine	218
GetRelatives Routine	220
Journal Routine.....	222
LinkMultiValue Routine	223
Message Routine	225
Parse Routine	227
PatternMatcher Routine	230
PropagateValue Routine	233
RankList Routine.....	235
ResetFrame Routine.....	236
SetMultiValue Routine.....	238
SetValue Routine.....	240
TestMultiValue Routine	241
Unify Routine	248
WriteTo Routine.....	251
3. Database Integration Topics	
Access String.....	253
Access String Specification	254
Arguments Overview.....	256
Atomic Retrieve.....	259
Atomic Write.....	261
Begin - (@BEGIN).....	263
Beginning Database Operations	264
Create New Record - (@FILL)	265
Create Object - (@FILL)	266
Cursor Slot Specification.....	266
Cursor - (@CURSOR).....	268
Database Interface Concepts	268
Database Editor Windows	273
Database Type - (@TYPE)	275
DBF3.....	277
Debugging Operations	278

Dynamic Values	281
End - (@END)	282
Ending Database Operations.....	283
Existence Filtering Operations	284
Field Name Specification	287
Fields List - (@FIELDS).....	287
File Retrieves - @F(...)	288
Formats	289
Forwarding Strategy - (@FWRD).....	290
Grouped Retrieve.....	291
Grouped Write.....	292
If Change Retrieves.....	294
If Change Writes.....	295
In List - (@ATOMS).....	296
INFORMIX	298
INGRES.....	304
Insert Only - (@FILL)	310
Interpretations - @V(...)	311
Left-Hand Side Retrieves	312
Left-Hand Side Writes	313
Link To - (@CREATE).....	313
Name - (@NAME)	314
New File - (@FILL)	315
NEXPERT Flat-File Formats	316
Object Names In Retrieve Operations.....	319
ORACLE.....	321
Order of Sources Retrieves	326
Order of Sources Writes	327
Properties List - (@PROPS)	328
Query (@QUERY).....	328
Query Language.....	329
Query Field in Retrieve Operations	335
Query Field in Write Operations	337
Record Specification for Writes.....	340
Records Filtering	345
Retrieve Operator.....	346
Retrieve Unknown - (@UNKNOWN).....	347
Retrieving from Databases	348
Return Errors	350
Right-Hand Side Retrieves	351
Right-Hand Side Writes	351
Sequential Retrieve	352
Sequential Write	354
Slot Specification for Retrieves.....	356
Slot Specification for Writes	358
Slots List - (@SLOTS)	360
Spreadsheets	361
SqlError - (@ERROR)	361
String to Numeric Conversion {x}	362
SYBASE.....	363
SYLK	369
WKS.....	370
Write Operator	371

Write Unknown - (@UNKNOWN).....	372
Writing to Databases	373
A. Database Integration Examples	
Example 1 - Grouped Write.....	375
Example 2 - Grouped Write with a Complex Name.....	378
Example 3 - Atomic Write.....	381
Example 4 - Grouped Retrieve	383
Example 5 - Grouped Retrieve with a Complex Name	387
Example 6 - Grouped Retrieve with Existence Filtering	390
Example 7 - Grouped Retrieve with Content Filtering.....	393
Example 8 - Atomic Retrieve.....	396
Example 9 - Sequential Retrieve	399
Example 10 - Sequential Retrieve with a Parameterized Query	402
Example 11 - Grouped Retrieve with a SQL Join	407
Index	409



Preface

Purpose of this Manual

This manual describes the application representation features available for use in your application development effort. Specifically, it addresses the implementation of these features in the Intelligent Rules Element shell, including their correct usage and syntax, where appropriate.

It also describes the Intelligent Rules Element Database Bridge. The database bridge is a link between your database and the Rules Element. Through this link, you can do two things: retrieve and write. You can retrieve data from your database and create objects in the Rules Element, and you can write Rules Element objects to your database.

Description

A wide variety of application representation features exist for use in the application development effort. These features include specific operators, functions, and execute routines, as well as conceptual features such as inference control, pattern matching, and dynamic objects. The application development environment of the Rules Element shell gives the developer easy access to these representation features through its use of popup menus and template-based editors.

Additionally, the Intelligent Rules Element database bridge lets you transfer data between external data sources and Rules Element's object representation. In many knowledge-based applications, the data is stored in an external file or database, where its format is very different from Rules Element's object representation. The object representation that includes classes, objects, properties, and slots provides a structure for data which the Rules Element reasons over. The database bridge transforms and translates the data between its external format (a file or database) and the Rules Element object representation.

Audience

This manual is the application developer's reference to locate specific topics during the application development effort. For example, developers can look-up the purpose of specific topics before implementing a feature in the application development environment of the Rules Element shell. Then during the implementation phase of the application, developers can locate examples in this manual to learn about syntax options.

Developers who want to embed Rules Element functionality directly into the code of another application should also refer to the API Reference. This alternative approach to applications design completely bypasses the graphical user interface and is therefore not addressed in the Language Reference.

How to Use this Manual

Developers can use this manual for reference purposes since the features appear in alphabetical order. Each feature has standard subtopics that give detailed information in the following categories: definition, syntax, arguments (if any), results, and examples. Additionally, each feature includes a listing of “related topics” that identify relevant information. The developer should always look-up the related topics in this manual before implementing the feature. The organization of this manual leaves the reading order up to the developer, but the related topics lists help to keep the topic investigation focused.

Chapter One “Application Development Features” describes the features that the developer uses to implement rule and object structures. A cross-section of the general representation features includes the following.

Test Operators	Determine the value of data or the logical state of subgoal hypotheses. Tests are used in the left-hand side (LHS) of rules.
Assignment Operators	Let you manipulate the value of slots in the application. Assignments can be made in the left-hand side or right-hand side of rules and methods.
Dynamic Objects Ops.	Let you manipulate objects and their links created during application processing (dynamically).
Interface Operators	Let you specify interactions with the outside world, including human operators, databases, user-written routines, or programs.
Inheritance Operators	Let you control both the strategy and the triggering of inheritance mechanisms.
Patterns	Let you perform queries on the object base. You can extract the list of objects that verify one or several conditions and then perform actions on the objects.
Formats	Let you specify how values should be output to the display, database, or data files. Also specifies how incoming text strings from the session control window, databases, data files, or the application programming interface (API) should be converted into the internal data types of the Rules Element.
Functions	Let you control both the strategy and the triggering of inheritance mechanisms.
Execute Routines	This category includes a full-range of pre-defined procedures for performing common or useful tasks. These routines are built into the system for use with the Execute operator.

Chapter Two, “Execute Library Routines” describes the functions in the execute library. They can be used like any user-defined execute routine in either conditions or actions of rules and methods. They can be divided up into several functional groups:

- Frame Operations This set of routines performs “crunching” operations on frames such as setting values, copying values, etc.
- Multi-Value Operations This set of routines performs operations on multi-values.
- Sorting and Comparison This set of routines performs operations on pattern matching lists.
- Session Control This set of routines controls the session and perform I/O.
- Utility Operations This set of routines performs useful tasks that extend application development.

Chapter Three, “Database Integration Topics” describes the key concepts, fundamental procedures, and general principles of the Intelligent Rules Element Database Bridge. This chapter includes topics from the following categories:

- Core Database Topics New users should read these first for more detailed information about the different ways the database bridge can be used and for detailed information about specific database types.
- Database Bridge Features Identifies features of the Rules Element Database Bridge that you can use to extend the database retrieve and write capabilities of your knowledge-base application.
- Rule Editor / Meta-Slot Editor Windows Lists topics specific to setting up database retrieve and write operations in a rule or method.
- Database Editor Windows Lets you find descriptions of the database editor windows’ various fields.
- Database Bridge Operations The topics in this list identify optional as well as required tasks of the retrieve and write operations. This information supplements the Database Editor Windows topics list.

This manual is a member of the document set. See “Related Manuals” for a complete list of prerequisite and corequisite manuals.

Organization

To locate specific features, look-up the features from one of the two chapters. All features appear in alphabetical order. The general table of contents identifies the complete features list and the index identifies more specific topics. This manual contains the following three chapters and one appendix:

Chapter One, “Application Development Features” describes the features that the developer uses to implement rule and object structures in the Intelligent Rules Element environment. All features appear in alphabetical order.

Chapter Two, “Execute Library Routines” explains how to use the special library of built-in routines the developer can invoke through the Execute operator. All routines appear in alphabetical order.

Chapter Three, “Database Integration Topics” gives information for key concepts, fundamental procedures, and general principles specific to building retrieve and write operations for a wide range of database types.

Appendix A, “Database Integration Examples” demonstrates the principles and operations of the Rules Element Database Bridge through specific examples.

Related Manuals

The following manuals contain information related to this Language Reference. Read prerequisite manuals before using this manual. Read corequisite manuals for background information as explained.

Prerequisite Manuals:

Getting Started

This manual gives an overview of the entire Rules Element shell, including the graphical user interface, the inference engine, and application representation features. Many of the design features described in the Reference Manual are first introduced in this manual.

User’s Guide

This manual gives general procedures for using the graphical user interface. Chapter Eight, “Application Data” of the User’s Guide shows how to perform Retrieve and Write operations. Additionally, Chapter Two, “Application Structure Implementation” of the User’s Guide gives useful information about rules and objects.

Corequisite Manuals:

Language Programmer’s Reference

This manual is required reading for developers who need an overview of the types of knowledge representation features available. The chapters describe the rule and object structures and control mechanisms that form

the basis of all Rules Element application development efforts. It also addresses the behavior of the inference engine.

The Bibliography, located in the Getting Started Manual, gives a complete list of manuals.

Users who received the Intelligent Rules Element packaged with other Neuron Data Elements, including the Open Interface Element and the Data Access Element, will have other documents in addition to the Intelligent Rules Element documents described above.

Application Development Features

This chapter describes the various application features of the Intelligent Rules Element.

ABS Function

Definition

The *ABS function* is used in expressions to find the absolute value of a floating point number. The expression can appear on the left-hand side or right-hand side of rules and methods.

Syntax

The function consists of the word `ABS` followed by a single argument in parentheses:

```
ABS(x)
```

Argument

The argument may be any expression yielding a numerical result. The expression may include patterns or interpretations.

Result

The function returns a floating point or integer result equal to the absolute value of the argument.

If the argument expression does not produce a numerical value, an error message is posted and the function result is `NOTKNOWN`.

Examples

The following examples illustrate the results of the `ABS` function:

```
ABS(98.6)      = 98.6  
ABS(-273.18)  = 273.18  
ABS(28)       = 28.0  
ABS(0.0)      = 0.0
```

Related Topics

Expressions

Floating Point Constants

Integer Constants

Patterns

Interpretations

ACOS Function

Definition

The *ACOS function* is used in expressions to find the arc-cosine of a floating point number. The expression can appear on the left-hand side or right-hand side of rules and methods.

Syntax

The function consists of the word `ACOS` followed by a single argument in parentheses:

```
ACOS ( x )
```

Argument

The argument may be any expression yielding a numerical result between `-1.0` and `1.0`. The expression may include patterns or interpretations.

If the value of the argument expression is an integer, it will be converted to an equivalent floating point value.

Result

The function returns a floating point result equal to the arc-cosine of the argument. The result is expressed in radians.

If the argument expression does not produce a numerical value, an error message is posted and the function result is `NOTKNOWN`.

Examples

The following examples illustrate the results of the `ACOS` function:

```
ACOS( 1.0 ) = 0.0
ACOS( 0.5 ) = 1.04 (= 3.14 / 3)
ACOS( 0.0 ) = 1.57 (= 3.14 / 2)
ACOS(-1.0) = 3.14
```

Related Topics

Expressions	Interpretations
Floating Point Constants	COS Function
Integer Constants	ASIN Function
Patterns	ATAN Function

Actions

Definition

An action expresses a unique operation to be performed by the system in a rule or method. Actions are normally considered consequences of a list of test conditions and would therefore appear on the right-hand side.

However, actions can also appear in the conditions list on the left-hand side. Actions that appear on the right-hand side can be divided into two sets: actions to be performed when the conditions on the left-hand side are

satisfied or actions to be performed when any single condition on the left-hand side fails to be satisfied.

Syntax

An action consists of an operator followed by one or two operands. The following operators can occur in both rules and methods:

Assign	Reset
CreateObject	Strategy
DeleteObject	Show
Retrieve	Execute
Write	LoadKB
SendMessage	UnloadKB

The following operators are valid in methods only:

InhMethod
Interrupt
NoInherit

The following operators are valid only in the right-hand side actions of Order of Sources methods:

AskQuestion	InhValueUp
Backward	RunTimeValue
InhValueDown	

The exact number and form of the operands varies from one operator to another; see the sections on individual operators for details.

Execution

Actions can appear on either the left-hand side (the “IF” section) or the right-hand side (the “ACTIONS” section) of rules and methods. On the right-hand side, actions can belong to one of two lists (the “Then” or “Else” lists). Which of these two actions list the system executes depends on the evaluation outcome of the rule or method. The execution of actions in rules and methods is as follows:

IF actions	Actions that appear in the left-hand side conditions list are executed sequentially in the order they appear. The “evaluation” result of an action is always set to <code>TRUE</code> .
Then actions	<p>Actions that appear in the “Then” list are executed sequentially in the order they appear, but the system must first determine that each left-hand side condition is <code>TRUE</code>. This is known as a positive rule or method evaluation.</p> <p>Note: The system automatically executes the Then actions defined for the method lacking a list of left-hand side conditions.</p>
Else actions	Actions that appear in the “Else” list are executed sequentially in the order they appear, but the system must first determine that one of the left-hand side conditions is <code>FALSE</code> . This is known as a negative rule or method evaluation.

The two part structure of the right-hand side allows actions to be executed whether or not the rule or method conditions succeed. This is equivalent to using two rules each with a different actions list to contend with the two

possible evaluation outcomes. The following two rules demonstrate how Else actions would be represented using only Then actions:

```
Rule1 IF Yes A => Hypo1
      THEN: Perform "true" actions list
Rule2 IF No Hypo1 = Hypo2
      THEN: Perform "false" actions list
```

Let's assume the system evaluates Rule1 first. After the evaluation of Rule1, the system forward chains to Rule2 due to the hypothesis test condition "No Hypo1" (called Hypothesis Forward). If Rule1's condition fails, then Hypo1 will be FALSE and Rule2's condition will evaluate to TRUE. Therefore, the failure of Rule1 ensures that the only actions list of Rule2 will be triggered. Or if Rule1 succeeds and its actions triggered, Rule2 will always fail. Thus only one set of actions can be triggered between these two rules. What took two rules can be accomplished more easily by including the Else actions list in a single rule as follows:

```
Rule1 IF Yes A => Hypo1
      Then Do: Perform "true" actions list
      Else Do: Perform "false" actions list
```

Forward Chaining

Depending on the inference strategy options currently in effect, the results of right-hand side actions may be forward-chained to the conditions of other rules that share the same data. If another rule shares the same data, its hypothesis is automatically placed on the agenda for consideration. This form of forward chaining is known as Forward Action-Effects. Methods are not affected by the results of actions because they do not have hypotheses to be considered for evaluation. However, actions in a method may forward-chain data to relevant rules. Action operators that will produce forward chaining include: `Assign`, `Retrieve` (from a database), and `Execute` (using an external routine).

Data that belongs to a private slot can not trigger action-effects since private slot data cannot appear in the conditions or actions of rules. Only data that belongs to public slots can trigger action-effects.

Depending on the inference strategy options currently in effect, only the results of the `Retrieve` and `Execute` actions triggered from rule or method conditions may be forward-chained. The `Assign` operator has no effect on forward chaining from the left-hand side. See the `Retrieve Operator` and `Execute Operator` topics for details.

Examples

The following example depicts the IF, THEN, ELSE construction that can be used in rules and methods.

```
IF Retrieve "data.nxp"
THEN Get information
ELSE Execute "Message" @TEXT="Error"
```

Related Topics

Rules	Agenda
Methods	Inference Strategy
Hypotheses	Inference Priority Attribute
Conditions	Forward Chaining
Slots	

Also see the sections on individual operators by name, as listed above.

Agenda

Definition

The *agenda* is the Rules Element's central control mechanism, which directs the course of its inference processing.

Form

The agenda is an ordered list of hypotheses pending investigation via inference processing. Notice that it is the hypotheses themselves that are placed on the agenda, not the rules that lead to them.

Operation

When the Knowcess command is issued to begin inference processing, the first hypothesis with the highest inference priority from the highest list on the agenda becomes the focus of attention, the object of active investigation by the Rules Element system. All rules leading to this hypothesis are investigated until its value is determined to be either `TRUE`, `FALSE`, or `NOTKNOWN`. Other hypotheses may be added to the agenda in the course of this inference process, as described under "Insertion," below.

As the value of each hypothesis is determined, it is removed from the agenda and the next hypothesis following it becomes the focus of attention. This process continues until all hypotheses have been processed and the agenda is empty, at which point the message End of Session is displayed in the session control panel of the Rules Element's main window.

Insertion

Although the user can explicitly place hypotheses on the agenda by selecting Suggest or related commands, the contents of the agenda are maintained automatically by the Rules Element and are not under the user's direct control. Hypotheses can be added to the agenda in any of the following ways:

1. Via an explicit suggestion by the user.
2. By backward chaining from the conditions of a rule already under investigation.
3. By forward chaining:
 - a. from the value of a hypothesis determined in the course of inference processing.
 - b. from a data value set in a rule by an action of some other rule.
 - c. from a data value set in a rule by an action of some method.
 - d. from a data value explicitly volunteered by the user.
4. Via a semantic gate from a rule previously investigated.
5. Via a context link from a hypothesis previously investigated.

Precedence

New hypotheses may be inserted in the agenda at any point, not just at the beginning or end. The list above shows the order of precedence: for example, hypotheses added to the agenda via semantic gates are placed after those reached via backward or forward chaining, but before those

reached via context links. Hypotheses added in the same way (for example, via semantic gates) are ordered according to their respective inference priorities or those of the rules leading to them.

Strategy

The ways in which hypotheses can be placed on the agenda are subject to the inference strategy currently in effect. The following strategy options apply:

- Forward confirmed hypotheses
- Forward rejected hypotheses
- Forward notknown hypotheses
- Forward Action-Effects (rules and methods)
- Forward through gates (rules only)

All of these options are normally enabled by default, but can be disabled if necessary. See the section “Inference Strategy” for more information.

Related Topics

Actions	Inference Slot Attribute
Conditions	Inference Strategy
Context Links	Rules
Hypotheses	Forward Chaining
Methods	Backward Chaining
Inference	Semantic Gates
Inference Priority Attribute	

For a thorough understanding of the Rules Element agenda mechanism, please refer to the Functional Description manual.

ASIN Function

Definition

The *ASIN function* is used in expressions to find the arc-sine of a floating point number. The expression can appear on the left-hand side or right-hand side of rules and methods.

Syntax

The function consists of the word `ASIN` followed by a single argument in parentheses:

```
ASIN(x)
```

Argument

The argument may be any expression yielding a numerical result between -1.0 and 1.0 . The expression may include patterns or interpretations.

If the value of the argument expression is an integer, it will be converted to an equivalent floating point value.

Result

The function returns a floating point result equal to the arc-sine of the argument. The result is expressed in radians.

If the argument expression does not produce a numerical value, an error message is posted and the function result is NOTKNOWN.

Examples

The following examples illustrate the results of the ASIN function:

```
ASIN( 0.0 ) = 0.0
ASIN( 0.5 ) = 0.52 (= 3.14 / 6)
ASIN( 1.0 ) = 1.57 (= 3.14 / 2)
ASIN(-1.0) = -1.57 (= -3.14 / 2)
```

Related Topics

Expressions

Floating Point Constants

Integer Constants

Patterns

Interpretations

SIN Function

ACOS Function

ATAN Function

AskQuestion Operator

Definition

The *AskQuestion operator* is used in the right-hand side actions of an Order of Sources method to prompt the user interactively for the value of a desired slot and test the answer.

Operand

The AskQuestion operator takes two operands:

- The first operand is a slot, commonly (but not necessarily) the one in whose Order of Sources method the AskQuestion operator appears (named in the Attach To field of the Method editor).
- The second operand is either a constant of the right type for the slot named as the first operand, or the special value NOTKNOWN.

Effect

The user is prompted, via the session control panel of the Rules Element main window, to supply a value for the slot to which this Order of Sources method belongs. If a prompt line attribute is defined for the slot, it is displayed in the window in place of the standard text.

After the value has been supplied, the slot named as the operator's first operand is tested for the value given by the second. If the two are unequal, the value supplied by the user is accepted and the method terminates; if they are equal, the value is rejected and execution continues with the next action in the Order of Sources method.

Example

The following actions, appearing in the `Order of Sources` method for an object's `cost` property, prompt the user to supply a value for that slot. Any known value is accepted; if the response is `NOTKNOWN`, the slot's value is instead inherited downward from a class or parent object:

```
AskQuestion SELF.cost NOTKNOWN
InhValueDown DEFAULT
```

Related Topics

Objects	Methods
Properties	Order of Sources Method
Actions	Prompt Line Attribute

Assign Operator

Definition

The `Assign` operator is used in the conditions and actions of rules and methods to assign a value to a variable.

Operands

The `Assign` operator takes two operands:

- The first operand can be a numeric constant, an arbitrary expression, a string, the special values `NOTKNOWN` or `UNKNOWN`, or a boolean constant (`TRUE` or `FALSE`) in the case where the second operand is a boolean variable.
- The second operand can be a slot, a list of slots specified by a pattern, or a boolean variable.

The operands may be of any type, but must both be of the same or compatible type. Any type of slot may be set to `NOTKNOWN` or `UNKNOWN`. Both operands may include patterns and interpretations. Note that a private slot used in the second operand is ignored unless the `Assign` operator appears in a method specifically triggered for the slot. See the description of Slots for more information about using private slots.

Effect

The value of the first operand is assigned to the slot named as the second. If both operands are identical, the effect is simply to force evaluation of the specified slot. For example, the following condition initiates backward chaining on the hypothesis "assigned" to itself.

```
Assign Hypo Hypo
```

If either or both operands include a pattern on the same class or object, the assignment is performed once for each object in the corresponding list. For example, the following condition assigns the product of the first operand to each object in the `Rect` class.

```
Assign <Rect>.length * <Rect>.width <Rect>.area
```

The condition with a pattern shown above, is equivalent to the following two conditions, assuming the Rect class contains two objects, Rect1 and Rect2.

```
Assign      Rect1.length * Rect1.width Rect1.area
Assign      Rect2.length * Rect2.width Rect2.area
```

Depending on the strategy options currently in effect, the new value of the slot assigned in an action of a rule or method may be forward-chained to other rules in which the slot appears in a condition (causing the hypotheses of those rules to be placed on the agenda for consideration). See the Forward Chaining section below for details. Also, the new value assignment may trigger the execution of the slot's If Change method, if one has been defined at the slot or parent slot level.

Forward Chaining

Right-hand side actions in rules and methods involving the `Assign` operator can forward chain the new value of the slot to other rules in which the slot appears in a condition (causing the hypotheses of those rules to be placed on the agenda for consideration). This form of forward chaining, known as Forward Action-Effects, is controlled first by a local strategy specific to the right-hand side Then and Else components of rules and methods. By default the local strategy is set to ON. If the local strategy is set to GLOBAL, the Rules Element defaults to the Rule Global forward action effects strategy in the Strategy Monitor window (from the Expert menu) until a `Strategy` operator overrides the global strategy at the local level.

Conditions of rules and methods involving the `Assign` operator are not able to initiate forward chaining. Values assigned by such a condition are never propagated forward to other rules, nor can such a condition be triggered by forward chaining from another rule or method.

Data that belongs to a private slot cannot trigger action-effects since private slot data cannot appear in the conditions or actions of rules. Only data that belongs to public slots can trigger action-effects.

Result

The result produced by the `Assign` operator is always `TRUE` unless the operands include a pattern with no matching values, in which case the result is `NOTKNOWN`.

Examples

The following are examples of conditions using the `Assign` operator:

```
Assign 3.14159          pi
Assign "Grumpy"        dwarf.name
Assign TRUE            Credit_approved
Assign FALSE          Credit_approved
Assign UNKNOWN        item.cost
Assign NOTKNOWN       switch_number
Assign DATE(1904,6,16) bloomsday
Assign item.count + 1  item.count
Assign rect_1.length * rect_1.width rect_1.area
```

Related Topics

Objects	String Constants
Rules	Boolean Constants
Methods	Patterns
If Change Method	Forward Chaining
Conditions	Inference Strategy
Actions	Strategy Operator
Data Types	Agenda
Expressions	Reserved Words
Slots	

ATAN Function

Definition

The *ATAN function* is used in expressions to find the arc-tangent of a floating point number. The expression can appear on the left-hand side or right-hand side of rules and methods.

Syntax

The function consists of the word `ATAN` followed by a single argument in parentheses:

```
ATAN(x)
```

Argument

The argument may be any expression yielding a numerical result. The expression may include patterns or interpretations.

If the value of the argument expression is an integer, it will be converted to an equivalent floating point value.

Result

The function returns a floating point result equal to the arc-tangent of the argument. The result is expressed in radians.

If the argument expression does not produce a numerical value, an error message is posted and the function result is `NOTKNOWN`.

Examples

The following examples illustrate the results of the `ATAN` function:

```
ATAN( 0.0) = 0.0
ATAN( 1.0) = 0.78 (= 3.14 / 4)
ATAN(999999) = 1.57 (= 3.14 / 2)
ATAN(-1.0) = -0.78 (= -3.14 / 4)
```

Related Topics

Expressions	Interpretations
Floating Point Constants	TAN Function
Integer Constants	ASIN Function
Patterns	ACOS Function

AVERAGE Function

Definition

The *AVERAGE function* is used in expressions to find the average of a set of numerical values. The expression can appear on the left-hand side or right-hand side of rules and methods.

Syntax

The function consists of the word `AVERAGE` followed by any number of arguments in parentheses:

```
AVERAGE(x1,x2,...,xn)
```

Arguments

Each argument may be any expression yielding a numerical or time-valued result. There may be either a list of arguments or a pattern matching list.

If some of the argument values are integers and some floating point, the integers will be converted to equivalent floating point values before computation.

Result

The function averages all the argument values and returns their arithmetic mean. For arguments that include patterns, it averages all values in the corresponding list.

Integer and floating point values may be mixed in the same average, but time values can be averaged only with each other. If numeric and time arguments are mixed, or if any argument is of another type, an error message is posted and the function result is `NOTKNOWN`.

Examples

The following examples illustrate the results of the `AVERAGE` function:

```
AVERAGE(365,240,577) = 394
AVERAGE(98.6,37.0,-273.18) = -29.85
AVERAGE(obj1.p,obj2.p,obj3.p) = 11.85
AVERAGE(TIME(8,4,23),TIME(3,6,11)) = TIME(5,35,17)
AVERAGE(123,"456") = NOTKNOWN
```

If class `Tank` has four instances with `capacity` values of 6.3, 14.5, 12.9, and 9.0, then

```
AVERAGE(<Tank>.capacity) = 10.67
```

Related Topics

Expressions

Data Types

TIME Function

Patterns

Interpretations

STDEV Function

DATE Function

VAR Function

Backward Chaining

Definition

Backward chaining is the process of determining the truth or falsity of a hypothesis by evaluating the rules that lead to it.

Invocation

Backward chaining is initiated by suggesting a hypothesis via any of the following commands:

- The `Suggest` command on the `Expert` menu.
- The `Suggest/Volunteer . . .` command on the `Expert` menu.
- The `Suggest . . .` command on the windows pop-up menu.
- The `Suggest Hypothesis` command on the `Rule editor` or `List of Rules` pop-up menu.
- The `Suggest` command on the `Rule Network`, `Object Network`, or `List of Hypotheses` pop-up menu.
- The `Suggest` command from the `Agenda Monitor`.

Backward chaining can also be initiated during runtime from the knowledge base itself:

- The `Assign Hypo Hypo` construct from a rule or method forces the evaluation of the hypothesis “assigned” to itself.
- The `Backward` operator may appear in an `Order of Sources` method as an action that backward chains to evaluate a hypothesis.

Each of these approaches places a hypothesis on the agenda for consideration. When the `Knowcess` command is issued to begin inference processing, the `Rules Element` will look for any inference rules leading to the designated hypothesis and begin evaluating them to determine whether the hypothesis is `TRUE` or `FALSE`.

Operation

Rules are considered one at a time in order of priority, as defined by their inference priorities. The results determine the value of the hypothesis, as follows:

- If any rule is found whose conditions are all `TRUE`, the hypothesis is set to `TRUE` and all of the rule’s actions are executed.
- Otherwise, if at least one rule has a condition that is `NOTKNOWN`, the hypothesis is set to `NOTKNOWN`.
- Otherwise, the hypothesis is set to `FALSE`.

The process terminates as soon as the value of the suggested hypothesis is determined, unless the strategy option `Exhaustive evaluation` is in effect; this option forces all rules leading to the suggested hypothesis to be evaluated, even after the value of the hypothesis has already been found.

In the course of evaluating a rule, hypotheses occurring in its conditions may in turn be placed on the agenda as subgoals, invoking the same reasoning process recursively to investigate all rules leading to those

hypotheses. Such backward chaining can continue recursively to any required depth.

Propagation

Depending on the global strategy options currently in effect, the results of the inference process described above may be propagated forward to other parts of the knowledge base, causing additional hypotheses to be placed on the agenda and additional rules to be evaluated. Strategy options relevant to this process include the following:

- Forward confirmed hypotheses
- Forward rejected hypotheses
- Forward notknown hypotheses
- Forward Action-Effects (rules and methods)
- Forward through gates (rules only)

See the section “Inference Strategy” for further details.

Related Topics

Agenda	Inference Priority Attribute
Boolean Constants	Inference Slot Attribute
Hypotheses	Inference Strategy
Rules	Forward Chaining
Inference	Exhaustive Evaluation
Backward operator	Assign operator

Backward Operator

Definition

The `Backward` operator is used in the actions list of an Order of Sources method to seek the value of a boolean slot by backward chaining to the inference rules in which it appears as a hypothesis.

Operand

The `Backward` operator is valid only in the THEN actions list on the right-hand side of an Order of Sources. The `Backward` operator takes one operand, which must be the boolean constant `TRUE`. The following is the only valid form for an action using the `Backward` operator:

```
Backward      TRUE
```

The `Attach To` field of the Method editor specifies the hypothesis to which the `Backward` operator applies.

The `Backward` operator cannot be used as an Order of Sources action for a private slot since private slot data cannot be a hypothesis. Only public slots can be hypotheses.

Effect

The `Backward` operator is meaningful only as an Order of Sources action for a boolean slot. If the slot appears as the hypothesis of one or more inference rules, it is placed on the agenda as a subgoal, causing its value to be sought by backward chaining on those rules. If there are two or more rules with the same hypothesis, they will be evaluated in the order specified by their inference priorities or inference slots.

Example

In the case of the boolean slot that is a hypothesis, the system triggers an available user-defined Order of Sources before it initiates backward chaining to obtain the value of the slot. To reincorporate the default behavior as part of a user-defined Order of Sources method, include the equivalent sequence of operators explicitly within the method:

```
InhMethod      DEFAULT
Backward       TRUE
InhValueDown   DEFAULT
InhValueUp     DEFAULT
AskQuestion    SELF          TRUE
```

Related Topics

Actions	Order of Sources Method
Agenda	Slot
Backward Chaining	Rules
Boolean Constants	Methods

Inference Priority Attribute

Inference Slot Attribute

Hypotheses

BOOL2STR Function

Definition

The `BOOL2STR` function is used in expressions to convert a boolean value to an equivalent character string. The expression can appear on the left-hand side or right-hand side of rules and methods.

Syntax

The function consists of the word `BOOL2STR` followed by one or two arguments in parentheses:

```
BOOL2STR (b)
BOOL2STR (b, f)
```

Argument

Each argument may be any expression yielding a result of the appropriate type:

- The first argument (`b`) is the boolean value to be converted.
- The optional second argument (`f`) is a string specifying the format under which the first argument is to be converted. See “Boolean Formats” for the syntax and meaning of this string.

The argument expressions may include patterns or interpretations.

Result

The function returns a string result representing the boolean value of argument `b`, converted according to format `f`. If no format argument is given, the default system format for booleans (defined in the `ckbres.format` module in the file `nrxrun.dat`) is used.

Examples

The following examples illustrate the results of the `BOOL2STR` function:

```

BOOL2STR ( FALSE )           = " FALSE "
BOOL2STR ( obj . p )        = " FALSE "
BOOL2STR ( FALSE , " Yup : Nope " ) = " Nope "

```

Related Topics

[Expressions](#)

[String Constants](#)

[Boolean Constants](#)

[Boolean Formats](#)

[Patterns](#)

[Interpretations](#)

[STR2BOOL Function](#)

Boolean Constants

Definition

A boolean constant is a sequence of characters that stand directly for a boolean (logical) value, representing the truth value of a condition or hypothesis or other boolean variables.

Values

There are two states that describe hypotheses and conditions: evaluated or unevaluated. Once the evaluation of a hypothesis or condition is complete, it resolves to one of the following boolean values.

```

TRUE
FALSE
NOTKNOWN

```

If a value has not yet been determined, the condition or hypothesis has the following boolean value.

```

UNKNOWN

```

An `UNKNOWN` value for a condition or hypothesis can be resolved to `TRUE`, `FALSE`, or `NOTKNOWN` as a result of further inference. A `NOTKNOWN` value can never be so resolved; its indeterminacy is an intrinsic condition of the problem itself and is usually volunteered by the user through the application interface. Both `UNKNOWN` and `NOTKNOWN` may be modified with the Assign operator.

Data Types

Although `NOTKNOWN` and `UNKNOWN` are applicable to boolean variables, slots of any data type may take these values. The values `TRUE` and `FALSE` are reserved for slots defined as type boolean. A fifth constant `KNOWN` is the counterpart of `UNKNOWN`, and may be assigned to slots of any data type.

All boolean constants are built into the Rules Element as reserved words

Related Topics

Data Types	Boolean Expressions
Identifiers	Reserved Words
Boolean Formats	Assign Operator

Boolean Expressions

Definition

A boolean expression represents a statement that when resolved returns a boolean result. It can make use of `AND`, `OR`, `NOT`, and embedded comparison operators.

Usage

The boolean expression is always used as an operand in a condition that returns a boolean result, such as a comparison or value test condition. There are two value test operators that return a boolean result:

`Yes`
`No`

There are six comparison operators that return a boolean result:

<code>=</code>	Equal
<code><></code>	Not equal
<code><</code>	Less than
<code><=</code>	Less than or equal
<code>></code>	Greater than
<code>>=</code>	Greater than or equal

Note: Comparison operators can also be embedded in the boolean expression itself.

Boolean Operators

Numeric or string values can be combined using the standard boolean operators when the result of the expression is a boolean value.

`AND` `OR` `NOT`

The Rules Element permits you to use these operators to combine values that individually evaluate to `TRUE`, `FALSE`, or `NOTKNOWN`. The result produced by the boolean expression depends on the evaluation of these values as described below.

Result

The result produced by a boolean expression is either TRUE, FALSE, or NOTKNOWN. The boolean operators provide the following results:

OR	TRUE	FALSE	NOTKNOWN
TRUE	TRUE	TRUE	TRUE
FALSE	TRUE	FALSE	NOTKNOWN
NOTKNOWN	TRUE	NOTKNOWN	NOTKNOWN

Boolean operator And provides the following results:

AND	TRUE	FALSE	NOTKNOWN
TRUE	TRUE	FALSE	NOTKNOWN
FALSE	FALSE	FALSE	NOTKNOWN
NOTKNOWN	NOTKNOWN	NOTKNOWN	NOTKNOWN

NOT:

NOT T == FALSE
 NOT F == TRUE
 NOT N == NOTKNOWN

Examples

The following is an example of a condition which tests a boolean expression:

Yes (a AND b) OR (NOT (c=1))

Related Topics

- | | |
|-------------------|----------------------|
| Conditions | Comparison Operators |
| Boolean Constants | No Operator |
| Expressions | Yes Operator |

Boolean Formats

Definition

A boolean format specifies the representation of a boolean value in text form for input and output purposes.

Syntax

This section defines the syntax of format elements for boolean-valued properties only. See the section titled "Formats" for the syntax of formats in general.

Like all formats, those for booleans may include strings of literal characters enclosed in double quotation marks (" . . . "), and may also include the wild-card character (*). Format elements beginning with an exclamation point (!) are ignored in database transactions; they are meaningful only for direct interaction with the user via the screen and keyboard.

Input

On input, each element in the format list is tried in order until one of them matches the input text. If no match is found, the input is rejected and an error message is displayed on the screen. The following conventions apply:

- Odd-numbered elements in the format list (the first, third, and so on) produce a `TRUE` result, even-numbered elements (the second, fourth, and so on) produce a `FALSE` result.
- Strings of literal characters enclosed in double quotation marks must match exactly, except that no distinction is made between upper- and lowercase letters.
- The wild-card character (*) matches any sequence of zero or more characters.

Output

On output, only the first two elements in the format list are used:

- The first format element is used for `TRUE` values, the second for `FALSE` values; any further elements in the list are ignored.
- Strings of literal characters enclosed in double quotation marks are reproduced exactly in the output.
- The wild-card character (*) is ignored on output.

Default

The default system format for booleans is defined in the `ckbres.format` module in the file `nrxrun.dat`. The standard default format is

```
True;False
```

Example

The following example illustrates the use of boolean formats:

Format: Yes;No;Y*;N*:@N=Maybe

Value	Output	Comments
<code>TRUE</code>	Yes	Uses first element
<code>FALSE</code>	No	Uses second element
<code>NOTKNOWN</code>	Maybe	Uses last (@N=) element

Input	Value	Comments
Yes	<code>TRUE</code>	Matches first element
no	<code>FALSE</code>	Case is irrelevant
Yup	<code>TRUE</code>	Matches third element
Nope	<code>FALSE</code>	Matches fourth element
NotKnown	<code>NOTKNOWN</code>	Reserved word
Maybe	<code>NOTKNOWN</code>	Matches last (@N=) element
Tru	<code>NOTKNOWN</code>	No match

Related Topics

Formats

Format Attribute

Boolean Constants

CEIL Function

Definition

The *CEIL function* is used in expressions to find the smallest whole number greater than a given floating point number. The expression can appear on the left-hand side or right-hand side of rules and methods.

Syntax

The function consists of the word `CEIL` followed by a single argument in parentheses:

```
CEIL(x)
```

Argument

The argument may be any expression yielding a floating point result. The expression may include patterns or interpretations.

Result

The function returns a floating point result equal to the smallest whole number greater than the argument. Notice that although the result is always a whole number, it is of type `FLOAT` rather than `INTEGER`. For negative arguments, the rounding is toward zero, rather than toward minus infinity.

If the argument expression does not produce a numerical value, an error message is posted and the function result is `NOTKNOWN`.

Examples

The following examples illustrate the results of the `CEIL` function:

```
CEIL(3.1416) = 4.0
CEIL(98.6)   = 99.0
CEIL(-273.18) = -273.0
CEIL(-9.9)   = -9.0
```

Related Topics

Expressions

Floating Point Constants

Integer Constants

Patterns

Interpretations

`FLOOR` Function

`ROUND` Function

CHARFIND Function

Definition

The *CHARFIND function* is used in expressions to search a character string for a specified character or characters. The expression can appear on the left-hand side or right-hand side of rules and methods.

Syntax

The function consists of the word `CHARFIND` followed by two arguments in parentheses:

```
CHARFIND(s1, s2)
```

Arguments

Each argument may be any expression yielding a string result:

- The first argument (*s1*) is the string to be searched.
- The second argument (*s2*) specifies the characters to search for.

The argument expressions may include patterns or interpretations.

Result

The function returns an integer result equal to the offset from the beginning of the first argument string (*s1*) to the first occurrence of any character from the second string (*s2*). The search is case sensitive, thus corresponding upper- and lowercase letters (such as `A` and `a`) are considered different for purposes of the search.

A result of 0 denotes the first character in string *s1* (no offset at all from the start of the string). If *s1* does not contain any of the characters in *s2*, the function result is equal to the length of string *s1*.

If either argument expression does not produce a string value, an error message is posted and the function result is `NOTKNOWN`.

Examples

The following examples illustrate the results of the `CHARFIND` function:

```
CHARFIND("SHAZAM!", "AEIOU") = 2
CHARFIND("SHAZAM!", "WXYZ") = 3
CHARFIND("SHAZAM!", "SPQR") = 0
CHARFIND("SHAZAM!", "LMNOP") = 5
CHARFIND("SHAZAM!", "aeiou") = 7
CHARFIND("SHAZAM!", "") = 7
CHARFIND("", "SHAZAM!") = 0
```

Related Topics

Expressions	SUBSTRING Function	STRUPPER Function
String Constants	STRLEN Function	STRLOWER Function
Patterns	STRCAT Function	STRFIND Function
Interpretations		

Classes

Definition

A *class* defines the common characteristics shared by a family of related objects.

Structure

Every class has a name, which must comply with the Rules Element's standard rules for a well-formed identifier. The class definition may include any number of *properties* to be inherited by the objects belonging to the class (called its *instances*). The class may also have any number of subclasses, which will likewise inherit its properties and pass them on in turn to their own instances. A given class may be a subclass of more than one other class (called its *superclasses*), just as a given object may be an instance of more than one class (called its including classes).

The class itself may associate a value with each property, independent of the property's value for any individual instance. Depending on the global and local inheritability settings currently in effect, the specific value of the property at the class level may or may not be inherited by instances or subclasses along with the property definition itself.

Inheritance

The default inheritability strategy allows both property definitions themselves and the specific values associated with them to be inherited downward from a class to its instances or subclasses. If necessary, these standard strategy settings can be changed from the Strategy Monitor window (from the Expert menu), the `Strategy` operator in a condition or action, or the Rules Elements application programming interface call `NXP_Strategy` to disable the inheritance of properties or their values or to permit upward as well as downward inheritance, from child to parent or parent to child. In addition, a class can override the global strategy settings by using the Meta-Slot editor to specify local inheritability attributes for individual slots associated with the class.

Creation

Classes can be created by several means:

- Interactively, via the `New` or `Copy` command in the Class editor.
- Implicitly, by using a previously undefined class name in a condition or action of a rule or method, or as a subclass of another class.
- Dynamically, through the Rules Element application programming interface (API).

Note: The system might display the class name enclosed between vertical bars (`| . . . |`) to distinguish it from an object name.

Deletion

Classes can be destroyed in either of two ways, depending on how they were originally created:

- Classes created interactively by the application developer, via the `Delete` command in the Class editor.
- Dynamically, through the Rules Element application programming interface.

Methods

Although a method is by definition triggered through a message sent directly to the object to which the method is attached, methods can be attached at the level of the class to govern the behavior of class instances. In the case where the object has no method attached, the system will try to use downward inheritance to obtain one. In a situation where the object belongs to multiple classes, each with its own method defined, then an `InhMethod` operator can be used to resolve the conflict by explicitly naming the parent class.

Related Topics

Objects

Properties

Identifiers

Rules

Meta-Slots

Inheritance

Slots

Methods

Dynamic Objects

Inheritability Strategy

Strategy Operator

InhMethod Operator

Retrieve Operator

Comment Attribute

Definition

The *comment attribute* is an arbitrary piece of text associated with a rule, method, or slot (a property of a class or object) to document its meaning or usage for the benefit of the application developer.

Syntax

The comment attribute may consist of any sequence of text characters, without restriction.

Effect

Comment attributes have no effect whatever on the operation of the system; their sole purpose is to help the application developer understand the structure and design of the knowledge base.

Creation

Comment attributes are specified or edited by typing into the box labeled `Comments` in the Meta-Slot editor (for an individual slot), the Rule editor (for a rule), or the Method editor (for a method).

Saving

Comment attributes are saved along with the knowledge base if the `Save Comments and Whys` option is chosen in the `Save Knowledge Base...` command.

Related Topics

Meta-Slots

Methods

Rules

Why Attribute

COMPARE Function

Definition

The *COMPARE function* is used in expressions to compare data values for equality or inequality. The expression can appear on the left-hand side or right-hand side of rules and methods.

Syntax

The function consists of the word `COMPARE` followed by two arguments in parentheses:

```
COMPARE ( x , y )
```

Arguments

Each argument may be any arbitrary expression. The expressions may include patterns or interpretations.

The argument values may be of any type, but the types must be comparable (either both the same or both numeric). If one is an integer and the other floating point, the integer will be converted to an equivalent floating point value before comparison.

Result

The function returns an integer result expressing the relation between the two argument values:

- If the first argument (*x*) is less than the second (*y*), the function result is -1.
- If the arguments are equal, the function result is 0.
- If *x* is greater than *y*, the function result is 1.

Integers and floating point values are compared numerically, strings lexically, and dates and times chronologically. In string comparisons, equivalent upper- and lowercase letters (such as `A` and `a`) are considered identical. In boolean comparisons, `TRUE` is considered greater than `FALSE`.

If the argument values are not of comparable types, the function result is `NOTKNOWN`.

Examples

The following examples illustrate the results of the `COMPARE` function:

```
COMPARE ( 365 , 240 )      = 1
COMPARE ( 98.6 , 98.6 )   = 0
COMPARE ( 12 , 12.0 )     = 0
COMPARE ( 12 , 12.3 )     = -1
COMPARE ( "Humpty" , "dumpty" ) = 1
COMPARE ( "boo" , "boojum" ) = -1
COMPARE ( "ABC" , "xyz" ) = -1
COMPARE ( "abc" , "XYZ" ) = -1
COMPARE ( "shazam!" , "SHAZAM!" ) = 0
COMPARE ( " " , "SHAZAM!" ) = -1
COMPARE ( DATE ( 1776 , 7 , 4 ) , DATE ( 1789 , 7 , 14 ) ) = 1
COMPARE ( TIME ( 8 , 4 , 23 ) , TIME ( 3 , 6 , 11 ) ) = 1
COMPARE ( TRUE , FALSE ) = 1
COMPARE ( 123 , "456" ) = NOTKNOWN
```

Related Topics

Expressions
Data Types

Patterns
Interpretations

Comparison Operators

Definition

The *comparison operators* are used in a rule's conditions to compare numerical values, dates, and times, as well as non-numeric values in the form of slots, strings, and booleans.

Operators

There are six comparison operators:

=	Equal
<>	Not equal
<	Less than
<=	Less than or equal
>	Greater than
>=	Greater than or equal

Note: For the operators = (equal) and <> (not equal) only, comparisons may be numeric or non-numeric as described below.

Numeric Operands

All comparison operators provide the means to compare two numeric values of the same or compatible type. To perform this type of comparison, the operators take the following two operands:

- The first operand may be any numeric constant, expression, or slot yielding an integer, floating point, date, or time value, and may include patterns.
- The second operand may be any numeric constant, expression, or slot yielding an integer, floating point, date, or time value, and may include patterns. It can also include KNOWN, NOTKNOWN, and UNKNOWN.

For the operators = (equal) and <> (not equal) only, the second operand may be a list of numbers separated by commas. Comparisons involving dates and times, must use the Date or Time function to yield a constant value.

Note that a private slot used in the second operand is ignored unless the comparison operator appears in a method specifically triggered for the slot. See the description of Slots for more information about using private slots.

Non-Numeric Operands

The operators = (equal) and <> (not equal) also allow comparison for equality between non-numeric values of the same or compatible type. To perform this type of comparison, the = and <> operators take the following two operands:

- The first operand can be either the name of a slot or a list of slots specified by a pattern.
- The second operand can be a list of one or more string or boolean constants separated by commas, or a single slot (patterns are not allowed). It can also include KNOWN, NOTKNOWN, and UNKNOWN.

Note: If the second operand is a slot, it must be of the same type defined for the first operand.

Result

The result produced by a comparison operator is TRUE, FALSE, or NOTKNOWN depending on whether the stated relation exists between the two operands. If the first operand includes a pattern, the condition tests whether at least one of the values in the corresponding list (for an existential pattern) or all of them (for a universal pattern) satisfy the given relation.

In the case of the operators = (equal) and <> (not equal), string constants listed in the second operand are recorded as possible values of the given slot, and will be presented as suggested options when requesting a value from the user for that slot.

Numeric Examples

The following are examples of conditions using the comparison operators to test numeric equality that involves variables, constants, and expressions:

```
> temperature 98.6
<= item_1.quantity * item_1.cost10000
<= <Item>.quantity * <Item>.cost10000
<= {Item}.quantity * {Item}.cost10000
= switch.number 8,14,22
> item_1.quantity * item_1.costmax_cost
```

Non-Numeric Examples

The following are examples of conditions using the = (equal) and <> (not equal) comparison operators to test strings and boolean constants:

```
= valve_1.pressure "increasing"
= valve_1.pressure "increasing", "stable"
= <Valve>.pressure KNOWN
= {Valve}.pressure UNKNOWN, NOTKNOWN
<> valve_1.pressure "increasing"
<> valve_1.pressure "increasing", "stable"
<> <Valve>.pressure KNOWN
<> {Valve}.pressure UNKNOWN, NOTKNOWN
```

Notice that the special values KNOWN, UNKNOWN, and NOTKNOWN are not written with string quotes (" ... ").

The following are examples of conditions using the = (equal) and <> (not equal) comparison operators to test the equality between two slots:

```
=      item_1.quantity      max_quantity
=      max_quantity        item_1.quantity
=      <Item>.quantity      max_quantity
=      {Item}.quantity      max_quantity
<>     item_1.quantity      max_quantity
<>     max_quantity        item_1.quantity
<>     <Item>.quantity      max_quantity
<>     {Item}.quantity      max_quantity
```

The following pattern matching statements are not valid comparisons and are illegal constructions due to the use of two dissimilar classes:

```
=      <itemA>.quantity      <itemB>.quantity
=      <itemA>.quantit      <itemB>.available_amount
```

Related Topics

Rules	Floating Point Constants
Conditions	Patterns
Slots	Expressions
Data Types	Boolean Expressions
Integer Constants	DATE Function
Boolean Constants	TIME Function
String Constants	

Conditions

Definition

A *condition* expresses a test to be performed on the left-hand side of a rule or method, helping to determine whether the rule or method is satisfied. Conditions in methods are optional.

Syntax

A condition consists of an operator followed by one or two operands. The possible operators are:

```
Yes      Write
No       CreateObject
=        DeleteObject
<>      Member
<        NotMember
<=      LoadKB
>       UnloadKB
>=     Reset
Assign   Show
Execute  Strategy
Retrieve endMessage
```

The exact number and form of the operands varies from one operator to another; see the sections on individual operators for details.

Rule Evaluation

The list of conditions within a rule is normally evaluated sequentially, in the order they appear in the rule definition; this evaluation order may be altered by the inference priorities of the data involved.

For the rule to be satisfied, all of its conditions must evaluate to TRUE. The conditions are thus implicitly linked by the logical “and” operator. To achieve the effect of a logical “or,” use separate rules leading to the same hypothesis.

The system executes one of two different lists of consequent actions (Then and Else) for the same rule depending on whether the rule is satisfied or not.

Method Evaluation

The list of conditions is optional for methods. If no conditions are present, the system automatically executes the Then actions list when the method itself is triggered. If method conditions are present, the system executes one of two different lists of consequent actions (Then and Else) depending on whether the method is satisfied or not.

For the method to be satisfied, all of its conditions must evaluate to TRUE. The conditions are thus implicitly linked by the logical “and” operator. To achieve the effect of a logical “or,” use backward chaining on separate rules.

If present, conditions within a method are always evaluated sequentially, in the order they appear in the method definition; unlike rule conditions this evaluation order is not altered by the inference priorities of the data involved.

Forward Chaining

Depending on the inference strategy options currently in effect, the evaluated data item or pattern in a condition may be forward-chained to the conditions of other rules that share the same data. In order for the hypothesis of another rule to be placed on the agenda for consideration, the forwarded data must make the condition of the target rule TRUE. This form of forward chaining is known as semantic gates. Methods are not affected by shared data because they do not have hypotheses to be considered for evaluation, nor can a condition in a method trigger forward chaining to another rule or method through a gate.

The system does not forward-chain the results of the Assign action triggered from the rule or method conditions list. However, depending on the inference strategy options currently in effect, the Retrieve and Execute actions triggered from the rule or method’s conditions list may be forward-chained. See the Retrieve Operator and Execute Operator topics for details.

Data that belongs to a private slot cannot trigger forward chaining since private slot data cannot appear in the conditions or actions of rules. Only data that belongs to public slots can trigger forward chaining.

Examples

The following examples illustrate conditions that can appear in a rule or method:

```
= car.color "blue", "red", "yellow"  
Yes Question_Answered OR Info_Retrieved
```

Related Topics

Rules	Comparison Operators
Methods	Boolean Constants
Hypotheses	Inference Priority Attribute
Actions	Semantic Gates
Slots	Inference Strategy
Forward Chaining	

Also see the sections on individual operators by name, as listed above.

Context Links

Definition

A *context link* (also called a *weak link*) is an explicit connection defined between two hypotheses to direct the course of the inference process. It is the only possible link between two knowledge islands.

Creation

Context links are always created interactively, via the New and Copy commands in the Context editor.

Deletion

Context links are always deleted interactively, via the Delete command in the Context editor.

Operation

Each time a hypothesis is investigated in the course of inference processing and its value (TRUE, FALSE, or NOTKNOWN) is determined, the Rules Element finds any other hypotheses that are connected to it via context links and places them on the agenda for later consideration. When these hypotheses come to the top of the agenda, their values in turn will be sought by backward chaining.

Asymmetry

Context links are one-directional: that is, a link from hypothesis A to hypothesis B does not also imply a link from B to A. For the connection to operate in both directions, two separate context links must be explicitly defined.

Precedence

Hypotheses generated as a result of context links have lower precedence (and consequently are placed lower on the agenda) than those generated either by backward chaining or via semantic gates. When several

hypotheses are placed on the agenda via context links, their precedence is determined according to their respective inference priorities.

Related Topics

Hypotheses	Backward Chaining
Rules	Forward Chaining
Boolean Constants	Inference Priority Attribute
Inference	Semantic Gates
Agenda	

COS Function

Definition

The *COS function* is used in expressions to find the cosine of a floating point number. The expression can appear on the left-hand side or right-hand side of rules and methods.

Syntax

The function consists of the word COS followed by a single argument in parentheses:

`COS (x)`

Argument

The argument may be any expression yielding a numerical result. The expression may include patterns or interpretations.

If the value of the argument expression is an integer, it will be converted to an equivalent floating point value.

Result

The function returns a floating point result equal to the cosine of the argument. The argument is assumed to be expressed in radians.

If the argument expression does not produce a numerical value, an error message is posted and the function result is NOTKNOWN.

Examples

The following examples illustrate the results of the COS function:

<code>COS (0 . 0)</code>	<code>= 1 . 0</code>
<code>COS (3 . 14 / 3)</code>	<code>= 0 . 5</code>
<code>COS (3 . 14 / 2)</code>	<code>= 0 . 0</code>
<code>COS (3 . 14)</code>	<code>= -1 . 0</code>

Related Topics

Expressions	SIN Function
Floating Point Constants	TAN Function
Integer Constants	ACOS Function
Patterns	COSH Function
Interpretations	

COSH Function

Definition

The *COSH function* is used in expressions to find the hyperbolic cosine of a floating point number. The expression can appear on the left-hand side or right-hand side of rules and methods.

Syntax

The function consists of the word `COSH` followed by a single argument in parentheses:

```
COSH( x )
```

Argument

The argument may be any expression yielding a numerical result. The expression may include patterns or interpretations.

If the value of the argument expression is an integer, it will be converted to an equivalent floating point value.

Result

The function returns a floating point result equal to the hyperbolic cosine of the argument.

If the argument expression does not produce a numerical value, an error message is posted and the function result is `NOTKNOWN`.

Examples

The following examples illustrate the results of the `COSH` function:

```
COSH( 0.0 )    = 1.0
COSH( 0.5 )    = 1.12
COSH( -0.5 )   = 1.12
COSH( 1.0 )    = 1.54
COSH( -1.0 )   = 1.54
```

Related Topics

Expressions	Interpretations
Floating Point Constants	<code>SINH</code> Function
Integer Constants	<code>TANH</code> Function
Patterns	<code>COS</code> Function

CreateObject Operator

Definition

The *CreateObject operator* is used in the conditions and actions of a rule or method to create dynamic objects in the course of inference processing, or to link existing objects to new including classes or parent objects.

The operator has an equivalent Rules Element application programming interface routine (`NXP_CreateObject`) and Rules Element Execute Library routine (`CreateObjects`).

Operands

The `CreateObject` operator takes one or two operands:

- The first operand is the name of an object or class and is usually specified as an interpretation of a slot.
- The optional second operand is a list of one or more class or object names separated by commas. Classes and objects may be mixed in the same list.

Either or both operands may include patterns or interpretations.

Effect

The object designated by the first operand is made an instance of each class and a component (subobject) of each object named in the second operand. If no object exists with the given name, a new one is created belonging to the given classes and parent objects.

If the first operand is a class rather than an object, it is made a subclass of each class named in the second operand.

If either operand includes a pattern, the operation applies separately to each object in the corresponding list.

Dynamic objects can have either public or private slots as determined by the parent object's slot attribute.

Any unknown name occurring in either operand will be created implicitly at compile time. Names enclosed within vertical bars (`| . . . |`) will automatically be created as classes; otherwise, the application developer will be prompted to identify the name as either a class or an object.

Dynamic objects and links created with the `CreateObject` operator can be deleted by the `DeleteObject` operator in the course of evaluating a rule or method. Additionally, dynamic objects and links exist only for the duration of the session in which they are created, and are automatically destroyed by the `Quit` or `Restart Session` commands.

Examples

If `whats_his_name` is a string slot whose value is `Rover` then the action

```
CreateObject      \whats_his_name\
```

creates a new object named `Rover`, belonging to no particular class or parent object.

If `Dog` is the name of a class and `my_pets` is an object, then

```
CreateObject      Rover      Dog,my_pets
```

makes the object `Rover` an instance of `Dog` and a component of `my_pets`.

```
CreateObject      |Poodle|      Dog
```

makes the class `Poodle` a subclass of `Dog`.

If `whats_his_name` is a string slot whose value is `Rover` then

```
CreateObject      'Good_Old_'\whats_his_name\      Dog
```

creates an object named `Good_Old_Rover` belonging to class `Dog`.

```
CreateObject<      my_pets>|      Animal|
```

makes every component of object `my_pets` an instance of a new class named `Animal`.

If `my_family` is an object,

```
CreateObject my_house< my_family>
```

links the existing object named `my_house` to every component of `my_family`.

```
CreateObject <my_pets> <Dog>
```

links every component of object `my_pets` to every instance of class `Dog`.

Related Topics

Objects	Actions
Dynamic Objects	Conditions
Classes	Slots
Rules	Patterns
Methods	Interpretations
DeleteObject Operator	

Data Types

Definition

Data types are the most basic units of information with which the Rules Element can work. There are six such types:

- Integer (32 bit whole numbers)
- Float (64 bit floating point numerical values)
- Boolean (logical values)
- String (sequences of text characters)
- Date (calendar dates and times of day)
- Time (intervals of duration)

There is also a seventh type named `Special`, representing the union of the other six: that is, a property of this type can take on values belonging to any of the other six elementary types. The use of this type is limited to the special property `Value`, used to carry the data value associated directly with an object itself. No other property can ever be defined to be of type `Special`.

Special values

All slots of a newly created object are initialized to the special value `UNKNOWN`, denoting a value that has not yet been determined. Another special value, `NOTKNOWN`, denotes a value that is definitively stated to be unspecified as one of the givens of the problem. An `UNKNOWN` value can be resolved to a specific data value as a result of further inference or computation. A `NOTKNOWN` value can never be so resolved; its indeterminacy is an intrinsic condition of the problem itself. Both `UNKNOWN` and `NOTKNOWN` may be modified with the `Assign` operator.

Related Topics

Objects	String Constants
Properties	DATE Function
Integer Constants	TIME Function
Floating Point Constants	Value Property
Boolean Constants	Assign Operator

Data Validation Attribute

Definition

Data Validation is used to predetermine an acceptable numeric range, list of strings, or more complex constraint for a slot or property whose value is determined at runtime.

Syntax

Data validation has three attributes. You can specify all or none as required for an individual slot. The attributes have the following syntax requirements:

Function	<p>You can specify a boolean expression to check the validity of the value entered for the slot. The slot must be referenced by <code>SELF</code>. Operators such as <code>AND</code>, <code>OR</code>, and <code>NOT</code> can be used, as well as any standard functions such as <code>RANDOM</code>. The functions <code>DATE</code> and <code>TIME</code> should be used to specify data and time values.</p> <p>Note: A compilation error will occur if you specify the slot by name; <code>SELF</code> must be used when referencing the slot displayed by the Meta-Slot editor.</p>
Execute	<p>You can specify an external routine installed through the Rules Element application programming interface call <code>NXP_SetHandler</code> to specify more complex constraints. The routine must return <code>TRUE</code> or <code>FALSE</code>.</p>
Error Help	<p>You can customize the alert dialog help string. It can be made dynamic by using the <code>@V()</code> and <code>@SELF</code> syntax. If no help string is specified, the system displays a default alert window with the options <code>ABORT</code>, <code>ALLOW</code>, and <code>RETRY</code>.</p>

Data validation expressions can include pattern matching in order to match values against a list. Examples of such a validation function include:

```
SELF.VALUE = <Class>.prop
SELF.item = <items>.name
```

This example requires `SELF.VALUE` to match at least one of the objects in the class specified with the property given. The `SELF` variable is useful when the data validation attribute is inherited by the children of the object whose slot includes the validation function. The system replaces the `SELF` variable with the name of the object which inherits the validation function.

The list generated by the existential or universal pattern used in a validation function cannot be reduced by further patterns since it is local to the data validation expression.

If you specify an external routine in the “Execute field,” the system will automatically pass the slot name, the proposed value, and the result if any of the evaluation of the “Function field” expression to the routine. In turn the routine will return its decision to accept or reject the proposed value for the slot.

Private slots can be the subject of a data validation test, but cannot be used in the validation of another slot. Public slots have no such restriction.

Creation

Data validation is specified via the `Data Validation` fields in the Meta-Slot editor in the case of the individual slot. Data validation can also be specified via the Property editor in the case of an individual property. Both editors provide the same attributes.

Default

By default data validation is disabled by a strategy at the global level. The strategy must be enabled in order for the system to process data validation expressions defined in the Meta-Slot or Property editors.

Strategy

You can enable or disable all data validation functions at two separate levels:

End User Validation	When the value of the slot for which a data validation function exists is solicited from the end user through a question window.
Engine Validation	When the value of the slot for which a data validation function exists is provided during the inferencing session by one of the assignment operators (Assign, Execute, and Retrieve).

Both types of data validation are normally disabled by default, but can be modified if necessary globally through the Strategy Monitor window (from the Expert menu) or locally through the `Strategy` operator in a rule or method. Both provide the following options:

OFF (default)	No data validation checking of the values entered.
ON/ACCEPT	Accept the value entered when the data validation expression contains a slot not yet evaluated.
OFF/REJECT	Reject the value entered when the data validation expression contains a slot not yet evaluated.

Operation

Data validation is either enabled or disabled as determined by the strategy currently in effect. If it is enabled and the system receives a value from the end user or determines a value through an Order of Sources for example, the inference engine processes the data validation attributes for the slot in question. If no data validation expression has been defined for the slot, the system will first try to inherit the data validation attributes of the slot’s

parent class or object and then try the property of the slot. Finally, if the system determines that an incorrect value has been supplied, an alert dialog with the default help string appears:

```
New value <value> for slot <slot> doesn't satisfy <test>.
```

You can customize the text of the alert dialog by using the `@V()` and `@SELF` syntax.

Inheritance

Inheritability of data validation attributes is controlled by the inference engine. The search for inheritable data validation attributes occurs from the more specific to the more general. If no data validation expression or execute routine has been defined for the slot, the system will try to inherit the data validation attributes of the slot's parent class or object. If none is available at the parent level; it will check at the property level.

Examples

The following example illustrates the data validation function:

```
SELF.quantity*Department.factor ≤ Department.threshold
```

Related Topics

Meta-Slots	Strategy operator
Properties	DATE Function
Patterns	TIME Function
Slots	SELF

Date Formats

Definition

A *date format* specifies the representation of a date value in text form for input and output purposes.

Syntax

This section defines the syntax of format elements for dates only. See the section titled “Formats” for the syntax of formats in general.

The following special characters are meaningful in date formats:

Y, y	Year field
M, m	Month or minute field
D, d	Day field
H, h	Hour field
S, s	Second field

Note: It is important to use spaces between the format characters. For example, “dd mm yy” is a valid format, whereas, “ddmmyy” is not.

The meaning and usage of these fields are discussed in the relevant sections below. Only the first element in the format list is used for output; any further elements are meaningful for input only.

Like all formats, those for dates may include strings of literal characters enclosed in double quotation marks (" . . . "), and may also include the wild-card character (*). Format elements beginning with an exclamation point (!) are ignored in database transactions; they are meaningful only for direct interaction with the user via the screen and keyboard.

Year

A series of *Ys* or *ys* denotes a year field. Upper- and lowercase letters may be used interchangeably; the distinction is irrelevant. The following forms are recognized:

Format	Example	Meaning
YY	84	Abbreviated year (2 digits)
YYYY	1984	Full year (4 digits)

Uppercase *Y* and lowercase *y* may be used interchangeably.

The abbreviated, two-digit form applies to twentieth-century years (1900–1999) only. On input, only one or two digits are accepted and are considered to be prefixed implicitly by 19: for example, the input value 84 is interpreted as the year 1984, and 4 as 1904. On output, twentieth-century years are automatically abbreviated to their last two digits, but years in other centuries are represented in full: for example, 1990 is represented as 90, but 1492 as 1492. A year field of any length other than two always denotes a full four-digit year number.

Month

A series of *Ms* or *ms* denotes a month field unless immediately preceded by an hour field, in which case it is interpreted as a minute instead (see "Minute," below). The following forms are recognized:

Format	Example	Meaning
MMMM	JANUARY	Full month name, all caps
Mmmm	January	Full month name, initial cap
mmmm	january	Full month name, all lowercase
MMM	JAN	Three-letter abbreviation, all caps
Mmm	Jan	Three-letter abbreviation, initial cap
mmm	jan	Three-letter abbreviation, all lowercase
mm	01	Two-digit month number
m	1	One- or two-digit month number

Uppercase *M* and lowercase *m* may be used interchangeably in the last two cases. In the last case, the month number is represented in the shortest form possible, one or two digits depending on the month.

Day

A series of `DS` or `ds` denotes a day field. The following forms are recognized:

Format	Example	Meaning
<code>DDDD</code>	MONDAY	Full weekday name, all caps
<code>Dddd</code>	Monday	Full weekday name, initial cap
<code>dddd</code>	monday	Full weekday name, all lowercase
<code>DDD</code>	MON	Three-letter abbreviation, all caps
<code>Ddd</code>	Mon	Three-letter abbreviation, initial cap
<code>ddd</code>	mon	Three-letter abbreviation, all lowercase
<code>dd</code>	01	Two-digit day of month
<code>d</code>	1	One- or two-digit day of month

The three- and four-letter forms represent the day of the week. These forms are invalid for input; on output, the weekday for a given date is computed automatically and formatted in the specified form.

The one- and two-letter forms represent the day of the month, and do not distinguish between uppercase `D` and lowercase `d`. In the one-letter case, the day number is represented in the shortest form possible, one or two digits as the case may be.

Hour

A series of `HS` or `hs` denotes an hour field. The following forms are recognized:

Format	Example	Meaning
<code>hh</code>	01	Two-digit hour number
<code>h</code>	1	One- or two-digit hour number

The distinction between uppercase `H` and lowercase `h` is irrelevant. In the one-letter case, the hour number is represented in the shortest form possible, one or two digits as the case may be.

Minute

A series of `MS` or `ms`, immediately preceded by an hour field, denotes a minute field. (If not preceded by an hour field, it is interpreted as a month instead; see “Month,” above.) The following forms are recognized:

Format	Example	Meaning
<code>mm</code>	01	Two-digit minute number
<code>m</code>	1	One- or two-digit minute number

The distinction between uppercase `M` and lowercase `m` is irrelevant. In the one-letter case, the minute number is represented in the shortest form possible, one or two digits as the case may be.

Second

A series of `SS` or `ss` denotes a second field. The following forms are recognized:

Format	Example	Meaning
<code>ss</code>	01	Two-digit second number
<code>s</code>	1	One- or two-digit second number

The distinction between uppercase *S* and lowercase *s* is irrelevant. In the one-letter case, the second number is represented in the shortest form possible, one or two digits as the case may be.

Examples

The format

```
Dddd, Mmmm d, yyyy " at " hh:mm:ss;mm-dd-yy hh:mm:ss
```

will output dates in the form

```
Thursday, December 18, 1984 at 13:43:07
```

and will accept them as input in the form

```
12-18-84 13:43:07
```

The format

```
DDD D MMM YY;mm/dd/yy
```

will output dates in the form

```
THU 18 DEC 84
```

and will accept them as input in the form

```
12/18/84
```

Default

The default system format for dates is defined in the `ckbres.format` module in the file `nrxrun.dat`. The standard default format is

```
Mmm dd yyyy hh:mm:ss;mm dd yy hh:mm:ss;Mmm dd yyyy; mm dd yy
```

This format will output dates in the form

```
Dec 18 1984 13:43:07
```

and will accept them as input in any of the forms

```
Dec 18 1984 13:43:07
```

```
12 18 84 13:43:07
```

```
Dec 18 1984
```

```
12 18 84
```

Related Topics

Formats

Format Attribute

DATE Function

TIME Function

Time Formats

DATE Function

Definition

A *date* is a Rules Element data value representing a calendar date, optionally also including a time of day. See also the `TIME Function` topic.

Syntax

A date constant can be specified in either of two formats, similar to those for times (see the TIME Function topic):

```
DATE(year, month, day)
DATE(year, month, day, hour, minute, second)
```

The parameters year, month, day, hour, minute, and second are integer values falling within the following ranges:

```
0 ≤ year      ≤ 32767
1 ≤ month     ≤ 12
1 ≤ day       ≤ 31
1 ≤ hour      ≤ 24
1 ≤ minute    ≤ 60
1 ≤ second    ≤ 60
```

For example,

```
DATE(1904,6,16)
```

denotes the date 16 June 1904, and

```
DATE(1981,6,8,21,8,46)
```

denotes 8 June 1981 at 9:08:46 p.m.

Expressions

Dates and times can be combined arithmetically in various ways. You can add or subtract two time intervals to produce a third interval representing their sum or difference, subtract two dates to find the interval between them, or add or subtract a date and a time to produce another date. You can also multiply or divide a time by a number (integer or floating point). In summary, here are the valid arithmetic operations on dates and times:

```
time + time      yields time
time - time      yields time
date - date      yields time
date + time      yields date
date - time      yields date
number * time    yields time
time * number    yields time
time / number    yields time
```

Related Topics

TIME Function

Data Types

Expressions

Date Formats

YEAR Function

MONTH Function

DAY Function

WEEKDAY Function

YEARDAY Function

NOW Function

DATE2FLOAT Function

Definition

The *DATE2FLOAT function* is used in expressions to convert a date to an equivalent floating point value. The expression can appear on the left-hand side or right-hand side of rules and methods.

Syntax

The function consists of the word `DATE2FLOAT` followed by a single argument in parentheses:

```
DATE2FLOAT(d)
```

Argument

The argument may be any expression yielding a date result. The expression may include patterns or interpretations.

Result

The function returns a floating point result representing the number of seconds from midnight, 1 January 1970, to the given date `d`. If the date is earlier than 1970, the result will be negative.

Examples

The following examples illustrate the results of the `DATE2FLOAT` function:

```
DATE2FLOAT( DATE(1981,6,8,21,8,46) )      = 360882526.0
DATE2FLOAT( DATE(1904,6,16) )            = -2068416000.0
DATE2FLOAT( "16 June 1904" )             = NOTKNOWN
```

Related Topics

Expressions

`DATE` Function

`TIME` Function

Patterns

Interpretations

`FLOAT2DATE` Function

`TIME2FLOAT` Function

`DATE2STR` Function

DATE2STR Function

Definition

The *DATE2STR* function is used in expressions to convert a date value to an equivalent character string. The expression can appear on the left-hand side or right-hand side of rules and methods.

Syntax

The function consists of the word `DATE2STR` followed by one or two arguments in parentheses:

```
DATE2STR(d)
DATE2STR(d, f)
```

Argument

Each argument may be any expression yielding a result of the appropriate type:

- The first argument (`d`) is the date to be converted.
- The optional second argument (`f`) is a string specifying the format under which the first argument is to be converted. See the Date Formats topic for the syntax and meaning of this string.

The argument expressions may include patterns or interpretations.

Result

The function returns a string result representing the date value of argument *d*, converted according to format *f*. If no format argument is given, the default system format for dates (defined in the `ckbres.format` module in the file `nrxrun.dat`) is used.

Examples

The following examples illustrate the results of the `DATE2STR` function:

```
DATE2STR( DATE(1904,6,16) ) = "Jun 16 1904 00:00:00"
DATE2STR( DATE(1904,6,16), "m/d/yy" ) = "6/16/04"
DATE2STR( DATE(1904,6,16), "Dddd, Mmmm dd, yyyy" ) =
    "Thursday, June 16, 1904"
```

Related Topics

Expressions

String Constants

DATE Function

TIME Function

Date Formats

Patterns

Interpretations

DATE2FLOAT Function

STR2DATE Function

DAY Function

Definition

The *DAY function* is used in expressions to extract the day field of a date or time. The expression can appear on the left-hand side or right-hand side of rules and methods.

Syntax

The function consists of the word `DAY` followed by a single argument in parentheses:

```
DAY( d )
```

Argument

The argument may be any expression yielding a date or time result. The expression may include patterns or interpretations.

Result

The function returns an integer result equal to the day field of the argument. For date arguments, the result ranges from 1 to 31.

If the argument expression does not produce a date or time value, an error message is posted and the function result is `NOTKNOWN`.

Examples

The following examples illustrate the results of the `DAY` function:

```
DAY( DATE( 1492, 10, 12 ) )           = 12
DAY( DATE( 1981, 6, 8, 21, 8, 46 ) ) = 8
DAY( TIME( 8, 4, 23 ) )              = 0
DAY( TIME( 3, 6, 11, 22, 34, 17 ) )  = 11
DAY( "October 12, 1492" )           = NOTKNOWN
```

Related Topics

Expressions

DATE Function

TIME Function

Patterns

Interpretations

YEAR Function

MONTH Function

HOUR Function

MINUTE Function

SECOND Function

WEEKDAY Function

YEARDAY Function

NOW Function

DeleteObject Operator

Definition

The *DeleteObject operator* is used in a condition or action of a rule or method to remove instances from a class or components from a parent object.

Operands

The `DeleteObject` operator takes one or two operands:

- The first operand is the name of an object or class.
- The second operand is a list of one or more class or object names separated by commas. Classes and objects may be mixed in the same list.

Either or both operands may include patterns or interpretations.

Effect

If there is one argument and the object is a dynamic object, then the object is deleted. Otherwise, the link is destroyed between the object designated by the first operand and each class or parent object named in the second operand. The object itself is not destroyed, only the link between it and the designated classes or parents.

If the first operand is a class rather than an object, it is removed as a subclass of each class named in the second operand.

If either operand includes a pattern, the operation applies separately to each object in the corresponding list.

Any unknown name occurring in either operand will be created implicitly when the rule is compiled. Names enclosed within vertical bars (`| . . . |`) will automatically be created as classes; otherwise, the application developer will be prompted to identify the name as either a class or an object.

Instance and component links destroyed with the `DeleteObject` operator are eliminated only for the duration of the session in which they are deleted. If the underlying object was created dynamically (with the `CreateObject` operator), it will automatically be destroyed by the `Quit` or `Restart Session` commands; if it was created explicitly (for example, via the `Object editor`), it will continue to exist and its original instance and component relationships will be restored by those commands.

Examples

If `Dog` is the name of a class and `Fido` and `my_pets` are objects, then

```
DeleteObject      Fido      Dog,my_pets
```

removes `Fido` as an instance of `Dog` and as a component of `my_pets`.

If `Poodle` is a subclass of `Dog`,

```
DeleteObject      Poodle     Dog
```

eliminates the subclass relationship.

If `whats_his_name` is a string slot whose value is `Rover`, then

```
DeleteObject '      Good_Old_'\whats_his_name\ Dog
```

removes the object named `Good_Old_Rover` from class `Dog`.

```
DeleteObject      <my_pets>      Animal
```

removes every component of `my_pets` from class `Animal`.

If `my_family` is an object,

```
DeleteObject my_house      <my_family>
```

destroys the links between the object `my_house` and every component of `my_family`.

```
DeleteObject<      my_pets>      <Dog>
```

destroys all links between the components of object `my_pets` and the instances of class `Dog`.

Related Topics

Objects

Dynamic Objects

Classes

Rules

Methods

Actions

Conditions

Patterns

Interpretations

CreateObject Operator

Dynamic Data Exchange

This topic addresses DDE calls. The Rules Element is shown both as a client and as a server for DDE conversations. It contains the following topics.

Introduction

Dynamic Data Exchange (DDE) is a Microsoft Windows communication protocol. Using DDE, a Windows application (the client) starts up a second Windows application (the server), passes data, uses the functions of the server, and calls for results. An application can be engaged in several DDE “conversations” at the same time, acting as the client in some and as the server in others.

DDE Conversations

The syntax of a DDE message is based on the following pattern:

```
Operation Topic Arguments
```

where:

Operation is either Request, Poke, or Execute.

Topic (of the conversation) depends on the application.

Example: it can be the name of a spreadsheet file if Excel is the server.

Arguments depends on the operation.

Rules Element-Based Application as a DDE Client

A Rules Element application is the client and initiates a DDE conversation with the server application. The Rules Element kernel currently supports three DDE calls in the Execute library: DDE_Poke, DDE_Request and DDE_Execute. The arguments to the DDE_execute call (@STRING and @ATOMID which are edited in the Execute Dialog) depend on the type of the call and are documented below.

Note: to copy the names, you can use the central column “Select Execute” pop-up menu in the Rule or Method editor of the Rules Element development environment.

Execute “DDE_Poke”

DDE_Poke copies a value from the Rules Element memory into the designed remote reference of the server application. The Atoms argument contains the data to be passed to the server. The String argument contains the names of the DDE Application, the Topic, and the remote reference, separated by spaces. The data to be passed can also be passed as a fourth argument in the string line.

Note: In the case of Excel, remote references should be indicated using the format R1C1 rather than A1.

Execute “DDE_Request”

DDE_Request copies a value from a designed remote reference into a Rules Element slot. The Atoms argument contains the slot where the value will be pasted. The String argument follows the same syntax as DDE_Poke. The remote reference argument can also be passed in the Atoms argument as the value of a second slot.

Execute “DDE_Execute”

DDE_Execute passes commands from a Rules Element application to the remote application. The Atoms argument does not carry information. The String argument contains the names of the DDE Application, the Topic, and the command string to be execute by the server, separated by spaces. The syntax of the command string depends on the server and is usually documented in the server manuals.

Rules Element-Based Application as a DDE server

When a Rules Element application is used as a server in a DDE conversation, the Rules Element will respond to Execute, Poke and Request messages from other client applications using the DDE protocol as published by Microsoft. The topic of the conversation must be “DDE”. How to generate those DDE messages will be described in your client application manuals.

In the case of an Execute message, the Rules Element will recognize the following commands (not case sensitive):

Command	Action	Syntax
EXE_clear	Clear All KB	EXE_clear()
EXE_load	Load KB	EXE_load(KBName)
EXE_restart	Restart	EXE_restart()
EXE_run	Knowcess	EXE_run()
EXE_suggest	Suggest	EXE_suggest (hypoName)
EXE_volunteer	Volunteer	EXE_volunteer (atomName, value)

When running the Rules Element as a DDE server, you might want to prevent the Rules Element from coming in front of your client window and getting the Windows input focus. This can be achieved by adding the following lines to the WIN.INI file in your Windows root directory

```
[Smartelt]
banner=off
```

Note: DDE initialization messages should be sent to the application called by the Rules Element. You might need to rename your Rules Element-based application to Intelligent Rules Element, if you want the client to start up the Rules Element-based application.

Excel Examples

Excel™ is a popular spreadsheet application (similar to Lotus 1-2-3) from Microsoft. Both the Rules Element applications and Excel support DDE. Two Excel examples, DATA and WEATHER, are included with the development kit which use DDE features. The examples are contained in the directory EXAMPLES\EXCEL

Excel version 5 Notes

Starting in Excel version 5, the REQUEST(B5, "advice.str") macro cannot be used to retrieve the string. Use instead DDESpy.exe to show that the value is correctly sent to Excel.

Also, with Excel 5, in order to execute an Excel macro remotely from within the Rules Element, you must name the macro in Excel using the option: Name Define from the Insert menu of Excel.

Rules Element as an Excel DDE client

In the example called Data, the Rules Element plays the client role in a DDE conversation. The Rules Element uses the functionality of Excel to place data in a cell, get data from another cell after an Excel calculation is remotely performed, and finally has Excel display a graph showing results of the previous operations.

To run the Data demonstration:

- Start Excel, close Sheet 1, and open DATA.XLW.
- Start the Rules Element, load DATA.TKB, and open the list of DATA.

- Minimize the Program Manager and arrange the windows so they can all be seen as on the next figure. Please note that the value in cell R3C3 is 1 and that the total of column y is 5.
- With the window focus on the Rules Element, do a Restart, Suggest, and Knowcness to see the Rules Element put value 5 in cell R3C3, get the new total of 12 in the slot total.num from cell R6C3, and display a graph with the updated value.

Note: It happens that a "DDE Execute failed" message appears and that the Excel icon or title bar blinks after doing the execute. With the current version of Excel 4.0, we are not getting an acknowledgment to the DDE Execute Operation from Excel, even though the command is correctly executed.

The Rules Element commands used in this example are described as follows:

```
Execute ("DDE_Poke") (@WAIT=TRUE;@ATOMID=content.num;@STRING="Excel DATA.XLS R3C3");
```

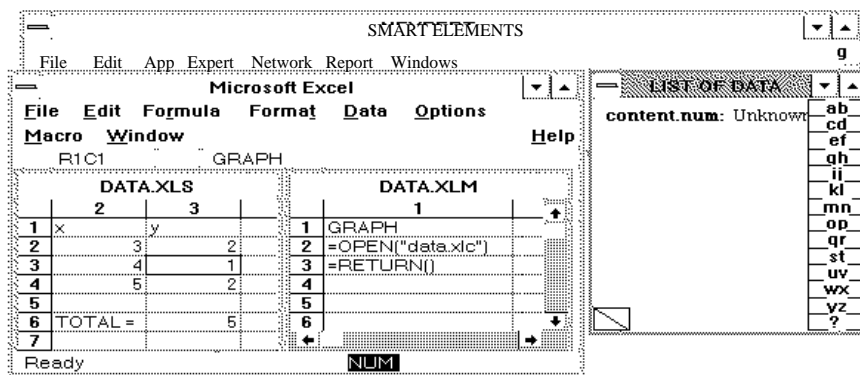
puts the string content.num (previously set to "8") into the cell designated in the STRING.

```
Execute ("DDE_Request") (@WAIT=TRUE;@ATOMID=total.num;@STRING="Excel DATA.XLS R6C3");
```

asks Excel for the content on cell R6C3 and places it in the slot total.num.

```
Execute ("DDE_Execute") (@WAIT=TRUE;@STRING="Excel DATA.XLM [RUN("R1C1")][BEEP()]");
```

tells Excel to run the macro contained in DATA.XLM.



Rules Element as an Excel DDE server

In the example called WEATHER, the Rules Element plays the server role in a DDE conversation. Excel volunteers data in the Rules Element, runs the inference engine (Knowcness), and writes the value of a Rules Element slot in a cell.

- Start the Rules Element.
- Start Excel, close Sheet 1, and open WEATHER.XLW. Then select cell B1 (or R1C2) in WEATHER.XLM to run the macro (select the option Run from the Macro Menu).

- The Excel sheet prompts you for several answers to questions. Successively answer:
OK to the Run window,
Rainy or Sunny to the weather condition and RETURN,

As a result of the DDE conversation, the cell R10C2 now displays the appropriate advice given by the Rules Element. You can see the data being displayed both in the Rules Element and in Excel by arranging the windows such as in the next figure. Check the option Display Formula of the Options menu in Excel on WEATHER.XLM macro sheet to see some of the commands that activate the Rules Element functions. Some of these commands are as follows:

```
=INITIATE("Intelligent Rules Element","DDE")
```

is the Excel macro to initiate a DDE conversation. Following DDE calls will refer to this conversation by its cell address, which is B5 in this case. The program the Rules Element needs to be already running. Note that the name is Intelligent Rules Element so that if your runtime is Rules Element-based only you will not need to change the Excel sheet.

```
= EXECUTE(B5, "EXE_clear()")
```

passes the command EXE_clear to be executed by the Rules Element. Consequently, the Rules Element will clear all databases that might be loaded at the time.

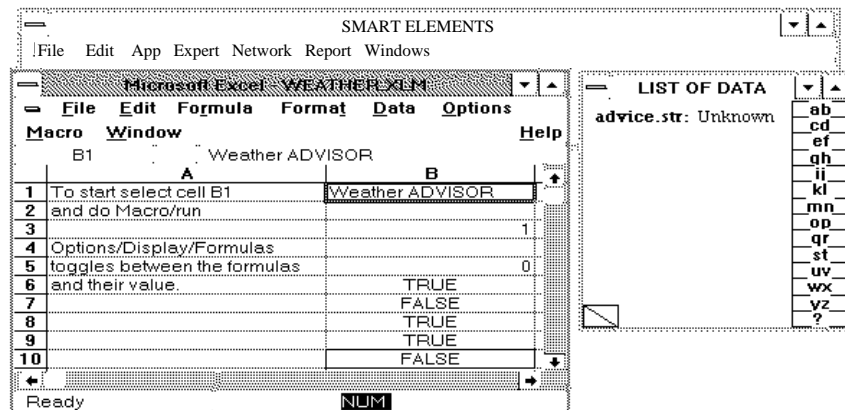
```
=REQUEST(B5,"advice.str")
```

is a DDE Excel Macro to request data from the Rules Element advice.str slot.

```
=TERMINATE(B5)
```

ends the DDE conversation.

The following figure shows the Rules Element as a server with Excel.



Dynamic Objects

Definition

A *dynamic object* is one that is created by a condition or action of a rule or method in the course of inference processing, rather than explicitly by the application developer.

Creation

Dynamic objects are created by executing the `CreateObject` operator in a condition or action of a rule or method. It also has an equivalent Rules Element application programming interface routine (`NXP_CreateObject`) and Rules Element Execute Library routine (`CreateObjects`).

The name of such an object need not be fixed in advance, but may be constructed dynamically from the value of a slot, using an interpretation: for example, if `whats_his_name` is a string slot whose value is `Rover`, then

```
CreateObject 'Good_Old_'\whats_his_name\ |Dog|
```

creates a dynamic object named `Good_Old_Rover` belonging to class `Dog`.

Dynamic objects can have either public or private slots as determined by the parent object's slot attribute.

Lifetime

Dynamic objects are temporary, existing only for the duration of the session in which they are created.

Display

When displayed on the screen (for example, in the Object editor, Object Network, or List of Objects), the name of a dynamic object is preceded by a plus sign in parentheses to indicate its dynamic nature:

```
(+)Good_Old_Rover
```

Deletion

The `DeleteObject` operator deletes dynamic objects. They are automatically deleted by the `Quit` or `Restart Session` command ending the session in which they are created.

Related Topics

Objects	Actions
Classes	Slots
Rules	<code>CreateObject</code> Operator
Methods	<code>DeleteObject</code> Operator
Conditions	Interpretations

Execute Operator

Definition

The *Execute operator* is used in rules and methods to invoke externally-written procedures or routines from the Rules Element library. See Chapter Two, "Execute Library Routines" for details about individual routines.

Operands

The `Execute` operator takes one or two operands:

- The first operand is a string constant or an interpretation which evaluates to a string constant (using the `@V(object.prop)` syntax) specifying the name of the external procedure to be invoked.
- The optional second operand consists of a series of execution parameters controlling the invocation of the procedure.

Parameters

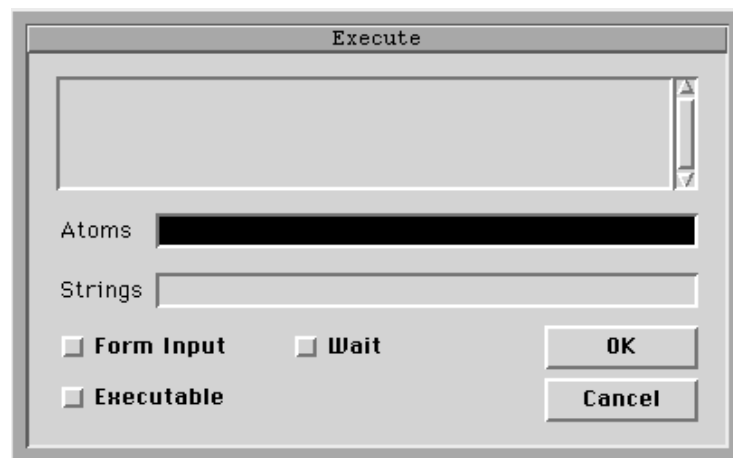
The second operand may include the following parameters:

<code>@STRING</code>	A string constant to be passed to the external procedure as an argument.
<code>@ATOMID</code>	A list of objects, slots, or classes to be passed to the external procedure as an argument.
<code>@TYPE=EXE</code>	External procedure is an executable file.
<code>@TYPE=FRM</code>	External procedure is a form.

See the *Intelligent Rules Element API Reference* for further details on the meaning and use of these parameters. Note that a private slot passed in the argument `@ATOMID` is ignored unless the `Execute` operator appears in a method specifically triggered for the slot. See the description of Slots for more information about using private slots.

Execute Dialog

When entering an `Execute` condition or action in the Rule editor or Method editor, clicking in the space for the second operand displays a special dialog box for specifying the execution parameters interactively, rather than by explicitly typing in the keywords listed above:



Effect

The external procedure named as the first operand is executed, using the argument values specified by the second operand.

Unless the parameter `@TYPE=EXE` is specified, the external procedure must previously have been installed as an execute handler via the Rules Element application programming interface routine `NXP_SetHandler` (described in the Intelligent Rules Element API Reference).

Result

When the `Execute` operator is used in a condition on the left-hand side of a rule, the return code of the executed procedure is checked; if it indicates success, the operator's result is set to `TRUE`, otherwise to `FALSE`.

Forward Chaining

Actions and conditions in rules and methods involving the `Execute` operator can forward-chain the new value of the slot to other rules in which the slot appears in a condition (causing the hypotheses of those rules to be placed on the agenda for consideration). In the case of the `Execute` operator, forward chaining is controlled by the global inference strategy setting from the Strategy Monitor window (from the Expert menu) and the local strategy which is always set to `CURRENT`.

Data that belongs to a private slot cannot trigger forward chaining since private slot data cannot appear in the conditions or actions of rules. Only data that belongs to public slots can trigger forward chaining.

Examples

The following are examples of conditions or actions using the `Execute` operator:

```
Execute      "flapdoodle"
Execute      "flapdoodle"@          TYPE=EXE;@STRING="mumble";
Execute"     @v(object.prop)"@      ATOMID=fee,|fie|,fo.fum;
```

Related Topics

Rules	Properties
Methods	Slots
Conditions	String Constants
Actions	Forward Chaining
Objects	Inference Strategy
Classes	Execute Routines

Also see the Intelligent Rules Element API Reference for more information on user-defined external procedures.

Refer to Chapter Two, "Execute Library Routines" for the complete list of available Rules Element routines.

Execute Routines

Definition

Rules Element *execute routines* are predefined external procedures for performing common or useful tasks, supplied with the system for use with the `Execute` operator.

Routines

The Rules Element run-time library includes the following routines:

Frame Operations

SetValue	GetRelatives
ResetFrame	PropagateValue
CopyFrame	CreateObjects

Multi-Value Operations

AtomNameValue	TestMultiValue
SetMultiValue	ComputeMultiValue
GetMultiValue	LinkMultiValue

Sorting and Comparison

RankList	PatternMatcher
GetListElem	Unify
FindListElem	

Session Control

ControlSession	Message
Journal	WriteTo

Utility Operations

AtomExist	FileExist
Parse	CreateReport

Each of these routines is fully described in its own section of this manual. Refer to Chapter Two, “Execute Library Routines.”

Invocation

Execute routines are invoked by using the `Execute` operator in a condition or action of a rule or method. The first operand to this operator is a string constant giving the name of the desired library routine; the second operand is a string consisting of a series of execution parameters to control the routine’s operation.

Parameters

Two standard execution parameters are used to specify the arguments of a library routine:

- The `@STRING` parameter passes a single string argument. If two or more such arguments are needed, they can be combined to form a multi-value and passed as a single argument; see the section “Multi-values” for more information.
- The `@ATOMID` parameter passes a list of objects, properties, or classes (typically specified via a pattern) for the library routine to operate on.

The specific usage of these parameters varies from one library routine to another, and is described in the section on each individual routine.

Note private slots must not be passed in the `@ATOMID` and `@STRING` parameter of the Execute routines. See the description of Slots for more information about using private slots.

Result

All execute routines return a result of `TRUE` if the call is successful, `FALSE` if an error occurs.

Dynamic Values

Individual atoms (objects and object properties) can be evaluated dynamically within the @STRING parameter by enclosing them within parentheses, preceded by the characters @V (for “value”). The atom’s current value will then be substituted into the @STRING parameter before execution.

For example, if Ducks.start contains the multi-value string Donald, Daisy and Ducks.more contains Huey, Dewey, Louie, then a condition or action of the form

```
Execute      "ComputeMultiValue"      @ATOMID=Ducks.start;
             STRING="@VALUE=
             @V(Ducks.more),@UNION,
             @RETURN=Ducks.all";
```

is equivalent to

```
Execute      "ComputeMultiValue"      @ATOMID=Ducks.start;
             @STRING="@VALUE=Huey,
             Dewey,Louie,@UNION,
             @RETURN=Ducks.all";
```

and will set the value of Ducks.all to the string Donald, Daisy, Huey, Dewey, Louie (the union of @Ducks.start and @Ducks.more).

Strategy Options

Many execute routines include an optional parameter named @STRAT as part of their @STRING parameter. This parameter is used to control the volunteering strategy for any value assignments made during the routine’s execution. It can be set to any of the following options:

SET	Store value immediately, but do not forward.
FWRD	Queue value for later forwarding if global strategy Forward Action-Effects is currently enabled.
SETFWRD	Combines both SET and FWRD options.

If no explicit @STRAT parameter is specified, the SET option is assumed by default.

Error Handling

Certain global flags can be used to control the handling of errors and tracing information by the built-in execute routines. All of these are boolean-valued objects whose Value properties contain the relevant flags:

SYS_ALERTFLAG	Report errors with alert handler
SYS_TRANSFLAG	Report errors in transcript
SYS_TRACEFLAG	Report trace messages in transcript
SYS_BEEPFLAG	Beep on error
SYS_STOPFLAGS	top session on error

These objects should be defined in a separate knowledge base so that they can be loaded in any session.

Related Topics

Conditions	Execute Operator
Actions	Patterns
Rules	Value Property
Methods	Multi-Values
Slots	Inference Strategy
String Constants	

Also see Chapter Two, “Execute Library Routines” for a detailed description of the routines.

EXP Function

Definition

The *EXP function* is used in expressions to find the natural (Napierian) exponential of a floating point number. The expression can appear on the left-hand side or right-hand side of rules and methods.

Syntax

The function consists of the word `EXP` followed by a single argument in parentheses:

```
EXP ( x )
```

Argument

The argument may be any expression yielding a numerical result. The expression may include patterns or interpretations.

If the value of the argument expression is an integer, it will be converted to an equivalent floating point value.

Result

The function returns a floating point result equal to e^x , the exponential of the argument to the Napierian base e ($= 2.71828$).

If the argument expression does not produce a numerical value, an error message is posted and the function result is `NOTKNOWN`.

Examples

The following examples illustrate the results of the `EXP` function:

```
EXP( 0.0 ) = 1.0
EXP( 0.5 ) = 1.64 (= SQRT(2.71))
EXP( 1.0 ) = 2.71
EXP(-1.0) = 0.36 (= 1 / 2.71)
```

Related Topics

Expressions	Patterns
Floating Point Constants	Interpretations
Integer Constants	LN Function

Expressions

Definition

An expression represents a computation to be performed on one or more elementary data values. Expressions can appear on the left-hand side or right-hand side of rules and methods. The system uses the expression result to complete the condition or action in which the expression appears.

Binary operators

Numerical (integer and floating point) values can be combined using the standard arithmetic operators:

+ - * /

The result of integer division is truncated toward zero. For example:

```
19 / 5 = 3
-19 / 5 = -3
19 / -5 = -3
-19 / -5 = 3
```

The arithmetic operators can also be applied in certain limited ways to date and time values; see the DATE Function and the TIME Function topics for details.

Boolean Operators

Numeric or string comparisons can be combined using the standard boolean operators when the result of the expression is a boolean value.

AND OR NOT

For example, the following expression has two requirements:

```
(x<10) AND (x>0)
```

Type conversion

If both operands to a binary operator are of the same type (integer or floating point), then the result is also of that type. If the operands are of different types, the integer operand is converted to floating point and the operation produces a floating point result. For example:

```
1 / 2 + 8 = 0 + 8 = 8
1 / 2 + 8.0 = 0 + 8.0 = 8.0
1.0 / 2 + 8 = 0.5 + 8 = 8.5
1 / 2.0 + 8 = 0.5 + 8 = 8.5
```

If an operand or function argument is not of the proper type or has the special value NOTKNOWN (denoting a value definitively stated to be unspecified), then the result of the expression is NOTKNOWN.

Precedence

The multiplication and division operators (* and /) take precedence over addition and subtraction (+ and -). Thus the expression

```
2 + 3 * 4
```

is evaluated as

```
2 + (3 * 4) = 14
```

rather than as

$$(2 + 3) * 4 = 20$$

Operators of the same precedence associate to the left: for example, the expression

$$3 * 7 / 9$$

is evaluated as

$$(3 * 7) / 9 = 21 / 9 = 2$$

rather than as

$$3 * (7 / 9) = 3 * 0 = 0$$

Functions

The following functions are built into the Rules Element and can be used freely in expressions:

Mathematical

ABS	ROUND	COMPARE
SIGN	CEIL	MAX
FLOOR	MOD	MIN
SIN	ASIN	SINH
COS	ACOS	COSH
TAN	ATAN	TANH
SQRT	EXP	RAND
POW	LN	RANDOM
LOG	RANDOMSEED	RANDOMMAX

Statistical

SUM	AVERAGE
PROD	VAR
STDEV	

Dates and Times

DATE	TIME	NOW
YEAR	HOUR	WEEKDAY
MONTH	MINUTE	YEARDAY
DAY	SECOND	

Strings and Lists

LENGTH	STRFIND	STRUPPER
STRLEN	SUBSTRING	STRLOWER
STRCAT	CHARFIND	

Conversion

STR2INT	STR2DATE	FLOAT2DATE
INT2STR	DATE2STR	DATE2FLOAT
STR2FLOAT	STR2TIME	FLOAT2TIME
FLOAT2STR	TIME2STR	TIME2FLOAT
STR2BOOL	FLOAT2INT	
BOOL2STR		

Related Topics

Data Types

DATE Function

TIME Function

Boolean Expressions

Also see the sections on individual functions by name, as listed above.

FLOAT2DATE Function

Definition

The *FLOAT2DATE function* is used in expressions to convert a floating point to an equivalent date value. The expression can appear on the left-hand side or right-hand side of rules and methods.

Syntax

The function consists of the word `FLOAT2DATE` followed by a single argument in parentheses:

```
FLOAT2DATE ( x )
```

Argument

The argument may be any expression yielding a numerical result. The expression may include patterns or interpretations.

If the value of the argument expression is an integer, it will be converted to an equivalent floating point value.

Result

The function returns a date result equivalent to the specified number of seconds (*x*) past midnight, 1 January 1970, rounded to the nearest second. If the argument value is negative, the result will be a date earlier than 1970.

Examples

The following examples illustrate the results of the `FLOAT2DATE` function:

```
FLOAT2DATE ( 250000000 )      = DATE ( 1977 , 12 , 3 , 12 , 26 , 40 )
FLOAT2DATE ( -777777777 . 7 ) = DATE ( 1945 , 5 , 9 , 22 , 37 , 2 )
FLOAT2DATE ( " 1234567 . 89 " ) = NOTKNOWN
```

Related Topics

Expressions

DATE Function
TIME Function
Patterns

Interpretations

DATE2FLOAT Function
FLOAT2TIME Function

FLOAT2INT Function

Definition

The *FLOAT2INT function* is used in expressions to convert a floating point number to an equivalent integer value. The expression can appear on the left-hand side or right-hand side of rules and methods.

Syntax

The function consists of the word `FLOAT2INT` followed by a single argument in parentheses:

```
FLOAT2INT ( x )
```

Argument

The argument may be any expression yielding a numerical result. The expression may include patterns or interpretations.

Result

The function returns an integer result which is equal to the integral portion of the argument. Thus if the argument is positive, it returns the `FLOOR` of the argument (as an integer), and if the argument is negative, it returns the `CEIL` of the argument (as an integer).

Examples

The following examples illustrate the results of the `FLOAT2INT` function:

```
FLOAT2INT(3.0)      = 3
FLOAT2INT(5.68)    = 5
FLOAT2INT(-4.54)   = -4
```

Related Topics

Expressions

Patterns

Ceil Function

Interpretations

Floor Function

FLOAT2STR Function

Definition

The *FLOAT2STR function* is used in expressions to convert a floating point value to an equivalent character string. The expression can appear on the left-hand side or right-hand side of rules and methods.

Syntax

The function consists of the word `FLOAT2STR` followed by one or two arguments in parentheses:

```
FLOAT2STR(x)
FLOAT2STR(x, f)
```

Argument

Each argument may be any expression yielding a result of the appropriate type:

- The first argument (*x*) is the floating point number to be converted.
- The optional second argument (*f*) is a string specifying the format under which the first argument is to be converted. See “Floating Point Formats” for the syntax and meaning of this string.

Argument *x* may also yield an integer value, which will first be converted to floating point and then to a string. The argument expressions may include patterns or interpretations.

Result

The function returns a string result representing the numeric value of argument *x*, converted according to format *f*. If no format argument is given, the default system format for floating point numbers (defined in the `ckbres.format` module in the file `nrxrun.dat`) is used.

Examples

The following examples illustrate the results of the `FLOAT2STR` function:

```
FLOAT2STR(98.6)           = "98.6"
FLOAT2STR(-273)          = "-273.0"
FLOAT2STR(1234.5, "k,u.0") = "1,234.5"
FLOAT2STR(0.9944, "%u.00\">%\"") = "99.44%"
```

Related Topics

Expressions	Floating Point Formats
String Constants	Patterns
Integer Constants	Interpretations
Floating Point Constants	STR2FLOAT Function

FLOAT2TIME Function

Definition

The *FLOAT2TIME* function is used in expressions to convert a floating point value to an equivalent time. The expression can appear on the left-hand side or right-hand side of rules and methods.

Syntax

The function consists of the word `FLOAT2TIME` followed by a single argument in parentheses:

```
FLOAT2TIME(x)
```

Argument

The argument may be any expression yielding a numerical result. The expression may include patterns or interpretations.

If the value of the argument expression is an integer, it will be converted to an equivalent floating point value.

Result

The function returns a time result equivalent to the specified number of seconds (*x*), rounded to the nearest second.

Examples

The following examples illustrate the results of the `FLOAT2TIME` function:

```
FLOAT2TIME(1234567.89)    = TIME(0,0,14,6,56,7)
FLOAT2TIME(-1234567.89)   = TIME(0,0,-14,-6,-56,-7)
FLOAT2TIME("1234567.89") = NOTKNOWN
```


Related Topics

Expressions

DATE Function

TIME Function

Patterns

Interpretations

TIME2FLOAT Function

FLOAT2DATE Function

Floating Point Constants

Definition

A *floating point constant* is a sequence of characters that stand directly for a floating point (real number) value.

Syntax

A floating point constant consists of one or more decimal digits (0–9), including a decimal point (.). It may optionally be preceded by a sign (+ or -) and/or followed by a decimal exponent. It may not include embedded spaces or commas. The decimal point is required in order to distinguish floating point from integer constants. The exponent, if present, is introduced by the letter E or e and may have an optional sign of its own, which is independent of the sign of the number itself.

The number after the letter E or e must be a constant (a slot is not allowed) and if used within a complex arithmetic expression, parentheses should be used:

`3.09E-3*POW(DensityConvtrtr.DensityIn,2)`

is ambiguous and should be written instead as:

`(3.09E-3)*(POW(DensityConvtrtr.DensityIn,2))`

Examples

The following are valid floating point constants:

2.718281828	38.0		
-273.18	38.		
+98.6	0.38		
6.02e23	.38		
+125e3	-125E+3	125e-5	-125E-5
1.25e+5	-1.25E5	+1.25E-3	-1.25e-3
125000.0	-125000.	.00125	-0.00125

The following are not:

xyz	Not a number
38	Integer, not floating point
62.5%	Contains an invalid character
\$1.98	Contains an invalid character
125 000.	Contains an embedded space
125,000.	Contains an embedded comma
125e2.5	Exponent not an integer

Related Topics

Data Types

Floating Point Formats

Integer Constants

Expressions

Floating Point Formats

Definition

A floating point format specifies the representation of a floating point value in text form for input and output purposes.

Syntax

This section defines the syntax of format elements for floating point properties only. See the section titled “Formats” for the syntax of formats in general.

The following special characters are meaningful in floating point formats:

k	Use next character as thousands separator
u	Suppress leading zeros
0	Placeholder for required digits
d	Placeholder for significant digits
%	Convert to percentage

The integral part of the number is represented by a series of zeros (0) specifying the minimum number of places preceding the decimal separator. The first nonzero character following this series defines the character to be used for the decimal separator itself, separating the integral and fractional parts. (This would normally be a period (.) in American or English usage, a comma (,) in some other countries.) The letter u in place of the zeros limits the integral part to the smallest number of digits actually needed to represent the given numerical value.

Following the decimal separator, the fractional part of the number is represented by a series of 0s followed by a series of ds, either or both of which may be empty. (Notice that all 0s must precede all ds.) The 0s denote required digits that must always be present; the ds denote optional additional digits to be included only if significant.

The letter k specifies that the next character following it is to be used as a *thousands separator*, dividing the integral part of the number into groups of three digits. (This would be a comma (,) in American or English usage, a period (.) or space in some other countries.) If the k is omitted, the integral part will be set as a solid series of digits, with no separators.

The percent sign (%) causes the number to be formatted in percentage form (for example, 0.25 as 25%).

Like all formats, those for floating point may include strings of literal characters enclosed in double quotation marks (" . . . "), and may also include the wild-card character (*). Format elements beginning with an exclamation point (!) are ignored in database transactions; they are meaningful only for direct interaction with the user via the screen and keyboard.

Input

On input, each element in the format list is tried in order until one of them matches the input text. If no match is found, the input is rejected and an error message is displayed on the screen. The following conventions apply:

- Odd-numbered elements in the format list (the first, third, and so on) produce a positive result, even-numbered elements (the second, fourth, and so on) produce a negative result.
- Input values of any length are recognized; placeholders (0 and d) used in the format to specify the number of digits before and after the decimal separator are ignored.
- The specified decimal separator is recognized as separating the integral and fractional parts of the input value.
- The thousands separator, if any, is optional on input.
- Strings of literal characters enclosed in double quotation marks must match exactly, except that no distinction is made between upper- and lowercase letters.
- The wild-card character (*) matches any sequence of zero or more characters.
- If the format includes a percent sign (%), the input supplied is interpreted as a percentage and is divided by 100 to arrive at the actual data value. (For example, an input value of 37.5 produces an actual data value of 0.375.)

Output

On output, only the first one or two elements in the format list are used:

- The first format element is used for positive and zero values, the second for negative values; any further elements in the list are ignored. If there is no second element, the first is used for all output values.
- A series of zeros (0) preceding the decimal separator in a format element specifies the minimum number of digits representing the integral part of the number. Numbers with integral parts shorter than this are padded with leading zeros; longer numbers are represented in full, using more than the specified number of digits.
- If the letter u precedes the decimal separator instead of a series of zeros, the integral part is represented in the minimum number of digits needed, with no leading zeros.
- A series of zeros (0) following the decimal separator in a format element specifies the minimum number of digits representing the fractional part of the number. Numbers with fractional parts shorter than this are padded with trailing zeros. Decimal places represented in the format by the letter d are included in the output only if they contain significant digits; trailing zeros in these positions are suppressed.
- If the fractional part exceeds the maximum length specified by the series of 0s and ds, it is truncated (not rounded) to the indicated number of digits.
- If a thousands separator is specified (introduced by the letter k), it is used to separate groups of three digits in the integral part of the number. No separator is used in the fractional part.

- Strings of literal characters enclosed in double quotation marks are reproduced exactly in the output.
- If the format includes a percent sign (%), the data value is interpreted as a percentage and is multiplied by 100 before being output. (For example, an actual data value of 0.375 produces an output value of 37.5.)
- The wild-card character (*) is ignored on output.

Default

The default system format for floating point is defined in the `ckbres.format` module in the file `nrxrun.dat`. The standard default format is

`0.0d`

denoting at least one digit before and after the decimal point and no thousands separator.

Examples

The following examples illustrate the use of floating point formats:

Format: "\$"k,0.00;"(\$"k,0.00)";u.d

Value	Output	Comments
1234.5	\$1,234.50	Positive uses first element
-1234.5	(\$1,234.50)	Negative uses second element
12.347	\$12.34	Truncated, not rounded

Input	Value	Comments
\$1,234.5	1234.5	Matches first element
\$1234.5	1234.5	Thousands separator optional
(\$1234.5)	-1234.5	Matches second element
1234.5	1234.5	Matches third element
-1234.5	-1234.5	Matches third element
1,234.5	NOTKNOWN	No match: first element has a dollar sign, third has no thousands separator
\$ 1234.5	NOTKNOWN	No match; space is significant

Format: %00.00d"%";;u.d*

Value	Output	Comments
0.062	06.20%	Converted to percentage
0.2533333	25.333%	Third decimal place is significant
-1.23	-123.00%	Exceeds integral length

Input	Value	Comments
6.20%	0.062	Matches first element
-123%	-1.23	Matches first element
6.20	6.2	Matches third element; no percentage conversion

Related Topics

Formats
Format Attribute

Floating Point Constants
Integer Formats

FLOOR Function

Definition

The *FLOOR function* is used in expressions to find the largest whole number less than a given floating point number. The expression can appear on the left-hand side or right-hand side of rules and methods.

Syntax

The function consists of the word `FLOOR` followed by a single argument in parentheses:

```
FLOOR(x)
```

Argument

The argument may be any expression yielding a floating point result. The expression may include patterns or interpretations.

Result

The function returns a floating point result equal to the largest whole number less than the argument. Notice that although the result is always a whole number, it is of type `FLOAT` rather than `INTEGER`. For negative arguments, the rounding is toward minus infinity, rather than toward zero.

If the argument expression does not produce a numerical value, an error message is posted and the function result is `NOTKNOWN`.

Examples

The following examples illustrate the results of the `FLOOR` function:

```
FLOOR(3.1416) = 3.0
FLOOR(98.6)   = 98.0
FLOOR(-273.18) = -274.0
FLOOR(-9.9)   = -10.0
```

Related Topics

Expressions
Floating Point Constants
Integer Constants
Round Function

Patterns
Interpretations
`CEIL` Function

Format Attribute

Definition

The *format attribute* associated with a property of a class or object specifies the representation of its value in text form for input and output purposes.

Syntax

The syntax for format attributes is described under “Formats” and in the sections on individual format types (such as “Integer Formats”).

Creation

The format attribute for a specific property of an individual class or object (public or private slot) is specified or edited by typing into the box labeled `Format` in the Meta-Slot editor. When specified, such an attribute overrides the format (if any) associated with the corresponding general, system-wide property that might have been specified in the Property editor.

Inheritance

Format attributes cannot be inherited.

Related Topics

Objects	Time Formats
Classes	Integer Formats
Properties	Floating Point Formats
Meta-Slots	Boolean Formats
Slots	String Formats
Formats	Date Formats

Formats

Definition

A format specifies the representation of a data value in text form for input and output purposes.

Creation

Formats can be specified either for a general, system-wide property or for a specific property of a given class or object (public or private slot). They are specified or edited by typing into the box labeled `Format` in the Property editor or the Meta-Slot editor, respectively.

Precedence

The applicable format for a given data item is determined according to the following order of precedence:

1. The format attribute (if any) associated with the specific data item (slot)
2. The format (if any) associated with the corresponding general property
3. The default system format for this data type (defined in the `ckbres.format` module in the file `nrxrun.dat`).

Syntax

A *format* consists of one or more individual *format elements* separated by semicolons (;):

```
element_1; element_2; element_3; . . .
```

The syntax for individual elements depends on the specific data type with which they are associated; see the sections on individual format types (such as “Integer Formats”) for details.

All format elements may include strings of literal characters enclosed in double quotation marks (" . . . "). Such quoted strings will be reproduced exactly on output and must be matched exactly on input. The quotes may be omitted if the literal characters do not form a meaningful combination within the format itself; this practice is discouraged, however, since the syntax of meaningful format elements may be subject to change in the future.

On input, an asterisk (*) in any format element acts as a “wild card” that will match any sequence of zero or more input characters. On output, it is simply ignored.

Format elements beginning with an exclamation point (!) are ignored in database transactions; they are meaningful only for direct interaction with the user via the screen and keyboard.

Special forms

In addition to those for specific data types, format elements may be defined for the special values UNKNOWN and NOTKNOWN. The syntax is as follows:

```
@U=format_string    for UNKNOWN values
@N=format_string    for NOTKNOWN values
```

For example, the format

```
@U="Who knows?";@N="¿Quién sabe?"
```

defines the strings `Who knows?` and `¿Quién sabe?` to stand for UNKNOWN and NOTKNOWN values, respectively. These strings will be used to represent the corresponding values on output and will be recognized as denoting them on input.

To avoid disturbing the sequence of odd and even format elements (see “Input,” below), such special UNKNOWN and NOTKNOWN format elements should always be placed at the end of the format list.

Input

On input, each element in the format list is tried in order until one of them matches the input text. If no match is found, the input is rejected and an error message is displayed on the screen.

For some data types, the identity of the matching format element may affect the resulting input value:

- For numerical (integer and floating point) data, odd-numbered elements (`element_1`, `element_3`, ...) produce a positive result, even-numbered elements (`element_2`, `element_4`, ...) produce a negative result.

- For boolean data, odd-numbered elements produce a `TRUE` result, even-numbered elements produce a `FALSE` result.
- For strings, dates, and times, the identity of the matching element does not affect the resulting value.

No distinction is made between upper- and lowercase letters in the input text: for example, the following are all considered identical:

```
february
February
FEBRUARY
fEbRuArY
```

If the user presses the space bar while entering input interactively from the keyboard, the Rules Element will attempt to complete the text automatically if it can be determined without ambiguity. For example, in entering the month field of a date, the letters `fe` will be expanded automatically to `February`; the letters `ju` will bring up a dialog window to choose between `June` and `July`.

Output

On output, only the first one or two elements in the format list are used:

- For numerical (integer and floating point) data, `element_1` is used for positive and zero values, `element_2` for negative. If `element_2` is not present, `element_1` is used for all values.
- For boolean data, `element_1` is used for `TRUE` values, `element_2` for `FALSE`.
- or strings, dates, and times, `element_1` is used for all values.

Any remaining elements in the format list are ignored.

Related Topics

Objects	Integer Formats
Classes	Floating Point Formats
Properties	Boolean Formats
Meta-Slots	String Formats
Slots	Date Formats
Time Formats	

Forward Chaining

Definition

Forward chaining is the process of propagating the values of public slots (objects and their properties) to the rules that refer to them, generating new hypotheses to be placed on the agenda for investigation. Methods are unable to be the target of forward chaining, but they have the ability to place the hypotheses of relevant rules on the agenda when public slots are involved. Private slots cannot initiate forward chaining since their value is accessible only by a method specifically triggered for the slot and will therefore not appear in any rule.

Invocation

Forward chaining is initiated explicitly by volunteering the value of a public slot via any of the following commands:

- The `Volunteer` command on the `Expert` menu.
- The `Suggest/Volunteer...` command on the `Expert` menu.
- The `Volunteer...` command on the windows pop-up menu.
- The `Volunteer` command on the `Rule Network`, `Object Network`, or `List of Data` pop-up menu.
- The `Volunteer/Modify` command on the `List of Objects` or `List of Classes` pop-up menu.

Each of these commands assigns new values to one or more slots, which can then forward-chain to any rules whose conditions refer to these slots.

Depending on the strategy options in effect, forward chaining can also occur implicitly, when values are assigned to hypotheses as a result of inference processing or to variables by the actions of rules and methods. The list of forward chaining inferencing processes includes:

- *Hypothesis Forward* occurs after the evaluation of a subgoal hypothesis (one that is tested in the condition of another hypothesis).
- *Forward Action-Effects* occurs after a rule or method action is executed and the result is shared with another rule condition.
- *Semantic Gates* occurs after data in a rule condition is evaluated that makes the condition of another rule `TRUE` upon propagation.

Operation

After assigning a new value to a public slot, the `Rules Element` searches for any existing rules whose conditions refer to that slot. The hypotheses of these rules are then placed on the agenda for consideration. When the `Knowcess` command is issued to begin inference processing, the values of these hypotheses will be sought by backward chaining. Notice that this can trigger the evaluation of all rules leading to the given hypotheses, not only those that refer to the originally volunteered slot.

Data that belongs to a private slot cannot trigger forward chaining since private slot data cannot appear in the conditions or actions of rules. Only data that belongs to public slots can trigger forward chaining.

Strategy

Forward chaining during the course of inference processing is subject to the global and local strategy options currently in effect. Options relevant to this process include the following:

- Forward confirmed hypotheses
- Forward rejected hypotheses
- Forward notknown hypotheses
- Forward through gates (rules only)
- Forward Action-Effects (rules and methods)

See the `Inference Strategy` topic for further details.

In addition to these global (system-wide) and local strategy options, forward chaining may be further restricted for individual rules by the values of their inference priorities; see the Inference Priority Attribute topic for more information.

Related Topics

Hypotheses	Agenda
Rules	Inference Strategy
Actions	Inference Priority Attribute
Slots	Assign Operator
Boolean Constants	Execute Operator
Inference	Retrieve Operator

HOURL Function

Definition

The *HOURL function* is used in expressions to extract the hour field of a date or time. The expression can appear on the left-hand side or right-hand side of rules and methods.

Syntax

The function consists of the word `HOURL` followed by a single argument in parentheses:

```
HOURL (d)
```

Argument

The argument may be any expression yielding a date or time result. The expression may include patterns or interpretations.

Result

The function returns an integer result equal to the hour field of the argument. For date arguments, the result ranges from 0 to 23.

If the argument expression does not produce a date or time value, an error message is posted and the function result is `NOTKNOWN`.

Examples

The following examples illustrate the results of the `HOURL` function:

```
HOURL (DATE (1492,10,12))           = 0
HOURL (DATE (1981,6,8,21,8,46))     = 21
HOURL (TIME (8,4,23))               = 8
HOURL (TIME (3,6,11,22,34,17))      = 22
HOURL ("October 12, 1492")          = NOTKNOWN
```

Related Topics

Expressions	DAY Function
DATE Function	MINUTE Function
TIME Function	SECOND Function
Patterns	WEEKDAY Function

Interpretations
YEAR Function
MONTH Function

YEARDAY Function
NOW Function

Hypotheses

Definition

A *hypothesis* is a slot with a boolean value, named on the right-hand side of a rule to specify the inference to be drawn from the rule's conditions.

Creation

New hypotheses may be created implicitly, by using a previously undefined name as the hypothesis of a rule in the Rule editor. On creation, such hypotheses are initialized to the special value `UNKNOWN`, meaning that their value is not yet determined; this setting may be resolved to `TRUE` or `FALSE` as a result of later processing. An existing boolean-valued object can also be made into a hypothesis by naming it as such in the hypothesis box of a rule.

Deletion

Hypotheses exist as objects in the Rules Element and can therefore be removed with the `Delete` command in the Object editor. Deleting a hypothesis also automatically deletes all rules leading to it if deletion is confirmed in the "Dependencies Warning Dialog Box."

Access

The current value of a hypothesis is denoted simply by the name of the hypothesis itself

`hypo_name`

(omitting the default `Value` property) or by an object or class name and a property name separated by a period

`object_name.prop_name`
`class_name.prop_name`

(if it is a property of some other object or class).

The value of the hypothesis may be set interactively via the Object editor, but it is normally computed by the Rules Element as a result of evaluating one or more rules. This can take place either through backward chaining (when the hypothesis itself is suggested as a goal to be inferred) or through forward chaining (when a data value in one of the rule's conditions is volunteered).

Related Topics

Objects
Properties
Rules
Conditions

Boolean Constants
Backward Chaining
Forward Chaining
Value Property

Identifiers

Definition

An *identifier* is a sequence of characters used as the name of a Rules Element atom, such as a rule, method, object, class, or property.

Syntax

An identifier consists of one or more letters (A–Z, a–z), digits (0–9), and underscores (`_`), beginning with a letter. It may be up to 255 characters; all characters are significant. Corresponding uppercase and lowercase letters are considered identical.

The underscore is a meaningful character and not just a null separator, which must be typed by the application developer during the editing session.

In some cases, the class name must be enclosed between vertical bars (`| . . . |`) to distinguish it from an object name.

Certain words, notably the names of Rules Element types, operators, functions, and special values, are reserved by the system and should not be used as ordinary identifiers. See the Reserved Words topic for a complete list.

Examples

The following are valid identifiers:

```
width          TOTAL
Finished       Btfsplk
taxRate        H2SO4
a_very_long_name_but_still_only_one_identifier
```

The following are not:

```
4to10         Doesn't begin with a letter.
_width        Doesn't begin with a letter.
Finished?     Contains an invalid character.
tax.rate      Contains an invalid character.
tax rate      More than one word.
Name          Reserved Word.
```

The following are all considered the same identifier:

```
taxrate       TAXRATE
taxRate       tAxRaTe
TaxRate
```

The following are different identifiers:

```
taxrate       tax_rate
```

Related Topics

Objects

Classes

PropertiesReserved Words

Expressions

If Change Method

Definition

An *If Change method* is an optional method that can be attached to a public or private slot (property associated with a class or object), defining the actions to be taken whenever the slot's value changes during the course of evaluating a rule or other method.

Structure

The method consists of most importantly a sequential list of actions, similar to those on the right-hand side of a rule. If desired, the If Change method can be structured exactly like a rule including a list of conditions on the left-hand side and two separate consequent lists of actions on the right-hand side. The conditions list is optional. Like all methods, the If Change method has no hypothesis component.

Creation

The If Change system method is specified via the Method editor. Creation begins by selecting the `Method` field and displaying the local popup menu for the edit line. Choose the `Select Method` option to view the selection dialog. Select the option `*IfChange` from the list (the asterisk in front of the name distinguishes it from user-defined methods). Or you can also type the name "IfChange" (one word) in the edit line for the `Method` field. The structure to which the method is attached is specified in the `Attach To` field. The structure you specify can be a slot, a class, or an object.

Invocation

In the case of public and private slots with an If Change method attached, the system automatically triggers the method whenever the value of the slot is changed during the inference process. A strategy option also permits slots that are reset to `UNKNOWN` to trigger the method. The If Change method actions list is executed in sequential order as soon as the value changes.

Optionally the method can be explicitly triggered by a `SendMessage` operator during the course of evaluating a rule or other method. This allows the application developer to trigger If Change actions instead of the inference engine. In the case of a class or object with an If Change attached, the `SendMessage` operator must be used in order to trigger the method, but it will no longer be dependent on the If Change strategy (and will actually be treated as a user-defined method by the inference engine).

If no explicit If Change method is specified at the level of the slot, a substitute method will be sought by downward inheritance from an including class, superclass, or parent object as directed by the inheritance strategy currently in effect. See the "Inheritance" section for details.

Operators

The following operators are valid in the conditions and actions of an If Change method:

Assign	Execute
SendMessage	LoadKB
CreateObject	UnloadKB
DeleteObject	Strategy

Retrieve	InhMethod
Write	NoInherit
Reset	Interrupt
Show	

Inheritance

If Change methods can only be inherited downward (from a class to its instances or subclasses, or from an object to its components), never upward. The search through the parent tree hierarchy is directed by the global inheritance strategy and can be class or object-first and depth or breadth-first. Any explicit If Change method defined at the level of the slot overrides this inheritance behavior; to reincorporate inheritance as part of such a method, include an explicit call to the `InhMethod` operator. To prevent the method from being inherited, change the `Public` option to `Private` in the Method editor.

When an inheritance conflict exists between two parent objects or classes at the same level, the application developer can use the `InhMethod` operator to override the default inheritance strategy by specifying the parent object to begin the search. When the inheritance conflict occurs between two slots at the same level, the application developer can set the inheritance priority of the slots to override the default inheritance strategy. If neither approach is used, by default the system chooses the method attached to the parent whose name appears first in alphabetic order. However, if the order is important, it is recommended that you force the method evaluation rather than rely on the default behavior.

Strategy

Automatic execution of If Change methods is normally enabled by default, but can be modified if necessary by changing the global inference strategy:

- Interactively through the Strategy Monitor window (from the Expert menu), by turning off the `If Change Actions` option (OFF).
- Dynamically during the course of inference processing itself, via the `Strategy` operator in a condition or action of a rule or method, using the `@CACTIONSON=OFF` setting.
- In addition to `ON` and `OFF`, a third option `ON/UNKNOWN` allows the system to trigger the If Change method not only when the value of the associated slot changes but also when it is reset to `UNKNOWN`. Unless this option is selected, values set to `UNKNOWN` will not trigger the If Change method.

Note: The `SendMessage` operator can be used to explicitly trigger an If Change method. The method triggered by the `SendMessage` operator is not affected by any of the strategy settings and will actually be treated as a user-defined method by the inference engine.

During the inferencing process the system first uses the `Strategy` operator setting to determine the global strategy, however, it is possible to invoke the Strategy Monitor window's If Change setting from the `Strategy` operator. This option is provided by the `CURRENT` setting in the `Strategy` operator argument dialog box.

Related Topics

Objects	Inheritance
Classes	Inheritance Strategy
Propertie	Inference
Actions	Inference Strategy
Rules	Strategy Operator
Methods	SendMessage Operator
Order of Sources Method	InhMethod Operator
Slots	

Also see the sections on individual operators by name, as listed above.

Inference

Definition

Inference is the process of reasoning by which the Rules Element determines the truth or falsity of hypotheses.

Techniques

The Rules Element uses two main inference techniques:

- *Backward chaining* begins with a hypothesis whose truth or falsity is to be determined and works backward to all rules leading to that hypothesis.
- *Forward chaining* begins with the value of a public slot and works forward to all rules whose conditions refer to that slot.

Either technique may generate further hypotheses or data values, continuing the inference process recursively to greater depths.

Invocation

Inference is initiated by *suggesting* one or more hypotheses to be investigated and/or *volunteering* one or more public slot values to be propagated. (See the Backward Chaining and Forward Chaining topics for more information.) These actions determine the *agenda* that will direct the course of the inference process; the contents of the agenda may be further modified dynamically in the course of processing. The Start With... Knowledge Base command on the Expert menu begins the inference process itself.

Private slots cannot initiate inferencing since their value is accessible only by a method specifically triggered for the slot and will therefore not appear in any rule.

Strategy

Various aspects of the inference process can be controlled or modified according to the global and local strategy options currently in effect; see the Inference Strategy topic for details.

Related Topics

Hypotheses	Slots
Rules	Backward Chaining
Boolean Constants	Forward Chaining
Object	Agenda
Properties	Inference Strategy

Inference Priority Attribute

Definition

An *inference priority* is a number that defines the priority and behavior of a hypothesis, rule, or data item during inference processing.

Value

The value of the inference priority must be an integer in the range ± 32000 .

Default

If no inference priority is explicitly defined, its value is 1 by default.

Effects

Inference priorities control the sequence of inference processing in the following ways:

- When two or more rules lead to the same suggested hypothesis, they are evaluated in the order of their inference priorities.
- For rules with equal inference priorities, the order of evaluation is determined by the highest inference priority among the data items referred to in each rule's conditions.
- Within a single rule, conditions are evaluated according to the highest inference priority among each condition's data items.
- When the Rules Element focuses on a new hypothesis within a particular inference agenda queue, it focuses on the hypothesis with the highest inference priority.

In each case, the order of evaluation is from highest inference priority to lowest.

Strategy Control

Certain specific ranges of inference priorities control the strategic behavior of a rule during inference processing. The effects of these special inference priorities are similar to disabling various strategy options (such as `Forward Action-Effects` or `Forward through gates`), but only for a single rule, rather than globally for the entire system. The following inference priorities apply to rules only; the negative values have no effect on hypotheses and data:

- **-32000 to -20001**: The rule is completely disabled and can never be reached during inference processing, either through forward or backward chaining.

- -20000 to -10001: The rule cannot be reached by any form of forward chaining, whether from the hypothesis of another rule, an action of a rule or method, a semantic gate, or a data value explicitly volunteered by the user. Such a rule can be reached only through backward chaining, when its hypothesis is suggested either explicitly (by the user) or implicitly (as a subgoal in the investigation of some other hypothesis).
- -10000 to -5001: The rule cannot be reached by forward chaining through a semantic gate.
- -5000 to -1001: The rule cannot be reached by forward chaining from an action in another rule or method.
- -1000 to 32000: The rule's inference behavior is unrestricted, subject only to the global strategy options currently in effect.

Creation

A rule's inference priority is specified via the `Inference Priority Number` box in the Rule editor; that of a slot (data item or hypothesis) is set by the `Inf Number` box in the Meta-Slot editor. The inference priority ranges described above are meaningful only when assigned to a rule through the Rule editor. Negative values assigned in the Meta-Slot editor have no effect on inferencing behavior resulting from the evaluation of data or hypotheses. The Meta-Slot editor in this case is used primarily to control the order of condition evaluation in a rule or the order of hypothesis evaluation.

Instead of a single fixed value, the inference priority can be calculated dynamically by designating an inference slot in the box labeled `Inf Priority Slot` in the Rule editor or `Inf Slot` in the Meta-Slot editor. If present, the value of the inference slot overrides that of the explicit inference priority.

Related Topics

Rules	Inference
Objects	Backward Chaining
Properties	Forward Chaining
Integer Constants	Inference Strategy
Hypotheses	Inference Slot Attribute
Conditions	Semantic Gates
Actions	Methods
Meta-Slots	

Inference Slot Attribute

Definition

An *inference slot* is a public slot whose value determines the priority and behavior of a hypothesis, rule, or data item during inference processing.

Value

The inference slot must be an integer public slot (a property of an object or class) with a value in the range ± 32000 . The negative range of values are useful on rules and otherwise have no effect on data or hypotheses. If it is the name of an object itself, its value is taken from the special `Value` property associated with the object.

Default

If no inference slot is defined or the defined slot's value is `UNKNOWN` or `NOTKNOWN`, the data or rule's explicit inference priority is used instead.

Operation

If an inference slot is specified, the value of the designated variable will be used in place of the explicit inference priority in determining the rule's or data item's inference priority and strategic behavior. This allows these attributes to be calculated dynamically at run time, rather than fixed unalterably in advance. See the Inference Priority Attribute topic for the specific meaning and effects of these numbers on rules. If the inference slot is `UNKNOWN`, the Rules Element will not try to determine its value (the Rules Element will use the inference priority or the default value).

Creation

The inference slot is specified via the box labeled `Inf Priority Slot` in the Rule editor or `Inf Slot` in the Meta-Slot editor. The slot name specified must be a public slot; a private slot cannot be used for this purpose.

Related Topics

Rules	Inference
Objects	Inference Priority Attribute
Propertie	Inference Strategy
Slot	Meta-Slots
Integer Constants	Value Property
Floating Point Constants	

Inference Strategy

Definition

Inference strategy controls the operation of the Rules Element's inference processing and the propagation of results from one inference rule to another.

Options

The following option selections are available for controlling the system's inference strategy. The keyword (preceded by an @ sign) following each strategy name is the abbreviation recorded in the text knowledge base:

- `Forward confirmed hypotheses (@PWTRUE)`: Any hypothesis which is in the context of a `TRUE` hypothesis will be put on the agenda for evaluation.

- Forward rejected hypotheses (@PWFALSE): Any hypothesis which is in the context of a FALSE hypothesis will be put on the agenda for evaluation.
- Forward notknown hypotheses (@PWNOTKNOWN): Any hypothesis which is in the context of a NOTKNOWN hypothesis will be put on the agenda for evaluation.
- Rule Global: Forward action-effects (@PFACTIONS): Any public slots whose values are changed by an Assign, Retrieve, or Execute operator involved in conditions or Then actions of a rule will be propagated forward to all rules that refer to them in their conditions. Note: The Assign operator never forwards actions from a condition, and the Retrieve and Execute operators only forward actions from a condition depending on the forwarding option selected. See each operator topic for details.
- Rule Else: Forward action-effects (@PFEACTIONS): Any public slots whose values are changed by an Assign, Retrieve, or Execute operator involved in the Else actions of a rule will be propagated forward to all rules that refer to them in their conditions.
- Method Global: Forward action-effects (@PFMACTIONS): Any public slots whose values are changed by an Assign, Retrieve, or Execute operator involved in conditions or Then actions of a method will be propagated forward to all rules that refer to them in their conditions. Note: The Assign operator never forwards actions from a condition, and the Retrieve and Execute operators only forward actions from a condition depending on the forwarding option selected. See each operator topic for details.
- Method Else: Forward action-effects (@PFMEACTIONS): Any public slots whose values are changed by an Assign, Retrieve, or Execute operator involved in the Else actions of a method will be propagated forward to all rules that refer to them in their conditions.
- Forward through gates (@PTGATES): After evaluating a rule, the inference process will propagate via semantic gates to any other rules with which it shares one or more public slots. The shared data item must make the condition of the target rule TRUE to be propagated.
- Exhaustive evaluation (@EXHBWRD): All rules leading to a suggested hypothesis will always be evaluated, even after the value of the hypothesis has already been determined by a previous rule.
- Enable order of sources (@SOURCESON): Order of Sources methods are in effect and will be executed when appropriate. Actions in Order of Sources methods may result in further inference processing depending on the current Forward Action-Effects strategy.
- Enable if change (@CACTIONSON): If Change methods are in effect and will be executed when appropriate. Actions in If Change methods may result in further inference processing depending on the current Forward Action-Effects strategy.
- User validation (@VALIDUSER): Enable validation of input solicited from the user before input is accepted for inferencing.
- Engine validation (@VALIDENGINE): Enable validation of input given by the system before input is accepted for inferencing (for example, from an Assign, Execute, or Retrieve).

Default

All inference strategy options listed above are normally enabled by default. The default settings can be modified interactively through the Strategy Monitor window (from the Expert menu) or during the course of evaluating a rule or method through the `Strategy` operator. See Global Control and Local Control below for details.

Global Control

The inference strategies listed above can be individually controlled through the Strategy Monitor window (from the Expert menu). The window has a list of checkboxes and menu buttons which determine whether a strategy is enabled or disabled. Clicking the mouse in any of the checkboxes toggles the corresponding strategy setting on or off. The darkened checkboxes show which inference options are currently enabled; unselected checkboxes are disabled. In the case of menu button controls, other options in addition to enabled and disabled are available from a menu that you display by clicking on the button. The currently displayed setting can be changed by selecting a new option from the list. During inferencing the settings may be changed interactively and placed into effect immediately.

Local Control

The system's inference strategy can be controlled locally during the course of inference processing via the `Strategy` operator in a condition or action of a rule or method. The `Strategy` operator selections override their corresponding global inference strategy, although the operator can default to the global strategy. The `Strategy` operator uses an arguments dialog box to control the inference strategies listed above with the following options:

ON	Enables the strategy until the next local strategy changes the setting.
OFF	Disables the strategy until the next local strategy changes the setting.
CURRENT	Invokes the corresponding Strategy Monitor window setting (from the Expert menu) until the next local strategy changes the setting.
GLOBAL	This option is used to synchronize control of the individual <u>Forward Action Effects</u> strategies (<code>@PFEACTIONS</code> , <code>@PFMACTIONS</code> , and <code>@PFMEACTIONS</code>). with the setting of "Rule Global Forward Action-Effects" (<code>@PFACTIONS</code>) that appears in the Strategy Monitor window. For instance, you can selectively enable or disable Else actions from a rule, or you can select the GLOBAL option so the strategy behaves exactly as the rule Then actions setting.

In addition to the local strategy options described here, the strategic behavior of individual rules and hypotheses can be controlled by using certain special values for their inference priorities: see the Inference Priority Attribute topic for details.

Related Topics

Hypotheses	Semantic Gates
Rules	Methods
Strategy	Order of Sources Method
Inference	If Change Method
Backward Chaining	Inference Priority
Forward Chaining	Strategy Operator

Inheritability Strategy

Definition

Inheritability strategy controls the inheritance of properties and their values from one object or class to another.

Variations

The following forms of inheritance can be controlled:

- Inheritance of property definitions between a class and its subclasses or instances.
- Inheritance of property definitions between an object and its components (subobjects).
- Inheritance of property values.

In each of these cases independently, inheritance may be permitted or forbidden in any direction or combination of directions:

- Downward (from class to subclass, class to instance, or parent object to component).
- Upward (from subclass to class, instance to class, or component to parent object).
- Both downward and upward.
- Neither downward nor upward.
- Private and public slots observe the same inheritability strategies. The private slot attribute controls the accessibility of the slot value and has nothing to do with inheritability.

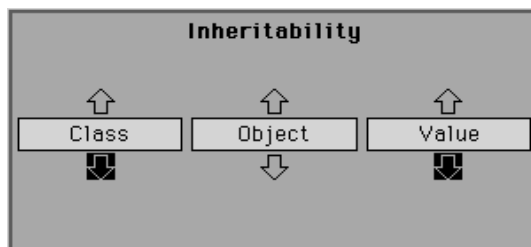
Default

The system's default inheritability strategy permits downward inheritance only, and only in the first and third cases listed above (property definitions from class to subclasses or instances, property values). Upward inheritance and inheritance between objects are disabled.

The default settings can be modified interactively through the Strategy Monitor window, during the course of evaluating a rule or method through the Strategy operator, or at the level of the individual slot. See Global Control and Local Control below for details.

Global Control

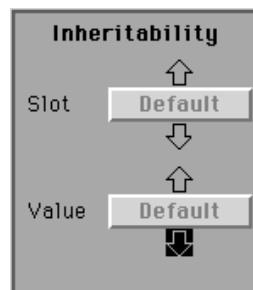
The global inheritability strategy in effect for the entire system can be set either with the Strategy Monitor window (from the Expert menu) or through the Strategy operator in a rule or method, using the options @INHCLASSDOWN, @INHCLASSUP, @INHOBJDOWN, @INHOBJUP, @INHVALDOWN, and @INHVALUP. In the Strategy Monitor window (from the Expert menu) the inheritability strategy is controlled by a diagram of the following form:



Clicking the mouse in any of the various arrows toggles the inheritability setting for the corresponding form of inheritance. Highlighted arrows show which inheritability options are currently enabled; those shown in the figure are for the standard default settings.

Local Control

The global inheritability strategy can be overridden in the case of individual slots through the Meta-Slot editor. In the Meta-Slot editor the inheritability strategy of the slot is controlled by a diagram of the following form:



In this case there are only two sets of arrows, controlling the inheritability of the slot itself and of its value, respectively. Clicking inside Default button sets the local inheritability strategy equal to the corresponding global strategy currently in effect.

The box labeled `Init Value` in the Meta-Slot editor lets you predetermine the value of the slot and specify whether or not it will be inheritable (`Public`) or not inheritable (`Private`). If an initial value is defined for the slot, it overrides the inheritability strategy currently in effect.

Related Topics

Objects	Meta-Slots
Classes	Inheritance
Properties	Inheritance Strategy
Rules	Strategy
Methods	Strategy Operator

Slots
Init Value Attribute

InhMethod Operator

Inheritance

Definition

Inheritance is a process by which characteristics of an object or class are propagated automatically to other, related objects or classes.

Variations

The following kinds of characteristics can be inherited:

- Property definitions
- Property values
- Slot accessible by rules (public slot) or method only (private slot)
- Data validation expression meta-slot or property attribute
- Prompt line meta-slot attribute
- Order of Sources and If Change methods
- Other user-defined methods.

Any of these characteristics can be inherited in the following ways:

- Between a class and its subclasses
- Between a class and its instances
- Between an object and its components (subobjects).

Direction

Inheritance can proceed in either of two directions (except methods and meta-slots):

- *Downward* (from class to subclass, class to instance, or parent object to component)
- *Upward* (from subclass to class, instance to class, or component to parent object)

Inheritance normally proceeds in the downward direction; upward inheritance is less common, but can be useful in some situations. Methods and meta-slot attributes can only be inherited downward, never upward.

Control

Inheritance takes place under the control of the global strategy settings currently in effect; see the sections “Inheritance Strategy” and “Inheritability Strategy” for details. The effects of these global settings are further modified by the local attributes (inheritance priority, inheritance slot, inheritance and inheritability attributes) associated with individual slots.

Additionally, specific inheritance behavior for individual slots can be defined via the following operators available through methods (the first two are valid only in Order of Sources methods):

InhValueUp
 InhValueDown
 InhMethod
 NoInherit

The meta-slot attributes, Data Validation and Prompt Line, are not under the control of the user; they are always inheritable in the downward direction. All other meta-slot attributes cannot be inherited including Format, Priorities, Question Window, and Why.

Related Topics

Objects	Inheritance Priority Attribute
Classes	Inheritance Slot Attribute
Properties	Inheritance Strategy
Meta-Slot	Inheritability Strategy
Methods	InhValueDown Operator
Order of Sources Method	InhValueUp Operator
If Change Method	InhMethod Operator
Strategy	NoInherit Operator

Inheritance Priority Attribute

Definition

An *inheritance priority* is a number that defines the inheritance priority of a slot.

Value

The value of the inheritance priority must be an integer in the range ± 32000 .

Default

If no inheritance priority is explicitly defined, its value is 1 by default.

Operation

In seeking an inherited value for a given slot, the Rules Element will give precedence to the candidate with the highest inheritance priority, subject to its global and local inheritability attributes. This principle applies at each ply of the search tree, under both depth-first and breadth-first inheritance strategies. The inheritance priority can therefore be used to resolve inheritance conflicts when a value is sought from multiple slots. Conflicts between methods attached to slots can also be resolved this way. (Note: Conflicts between methods attached to classes, objects, or properties must be resolved through the `InhMethod` operator.)

Creation

The inheritance priority is specified via the `Inh Number` box in the Meta-Slot editor. Instead of a single fixed value, the inheritance priority can be calculated dynamically by designating an inheritance slot in the box labeled `Inh Slot`. If present, the value of the inheritance slot overrides that of the explicit inheritance priority.

Related Topics

Objects	Inheritance
Classes	Inheritance Slot Attribute
Properties	Inheritance Strategy
Integer Constants	Inheritability Strategy
Meta-Slots	Methods

Inheritance Slot Attribute

Definition

An *inheritance slot* is a public slot whose value determines the inheritance priority of a slot.

Value

The inheritance slot must be an integer public slot (a property associated with an object or class) with a value in the range ± 32000 . If it is the name of an object itself, its value is taken from the special `value` property associated with the object.

Default

If no inheritance slot is defined, the system will use the explicit inheritance priority of the slot whose value is being sought. See the Inheritance Priority Attribute topic for details.

Operation

If an inheritance slot is specified, the value of the designated variable will be used in place of the explicit inheritance priority in determining the priority with which the slot's value can be inherited by other objects or classes. This allows the inheritance priority to be calculated dynamically at run time, rather than fixed unalterably in advance. If the inheritance slot is `UNKNOWN`, the Rules Element will not try to determine its value (the Rules Element will use the inheritance priority or the default value).

In seeking an inherited value for a given slot, the Rules Element will give precedence to the candidate with the highest inheritance priority, subject to its local inheritability attributes and the global inheritability strategy currently in effect. This principle applies at each ply of the search tree, under both depth-first and breadth-first inheritance strategies. The inheritance slot can therefore be used to resolve inheritance conflicts when a value is sought from multiple slots. Conflicts between methods attached to slots can also be resolved this way. (Note: Conflicts between methods attached to classes, objects, or properties must be resolved through the `InhMethod` operator.)

Creation

The inheritance slot is specified by typing the name of the slot into the `InhSlot` box in the Meta-Slot editor. The slot name specified must be a public slot; a private slot cannot be used for this purpose.

Related Topics

Objects	Floating Point Constants
Classes	Inheritance
Properties	Inheritance Strategy
Slots	Inheritability Strategy
Meta-Slots	Inheritance Priority Attribute
Integer Constants	Value Property

Inheritance Strategy

Definition

Inheritance strategy controls the order in which a slot value or method is inherited from its including classes and parent objects. If the same property can be inherited from more than one source, the strategy determines which source will actually be used.

Variations

The search for an inherited value of a given property can be conducted in either of two ways:

- *Class-first*, examining the classes to which the object belongs before the parent objects of which it is a component.
- *Object-first*, examining parent objects before classes.

In either case, the search can proceed in either of two orders:

- *Breadth-first*, examining all of the object's immediate classes or parent objects before any of their own more remote ancestors.
- *Depth-first*, examining each complete chain of superclasses or superobjects to its full depth before moving on to the next.

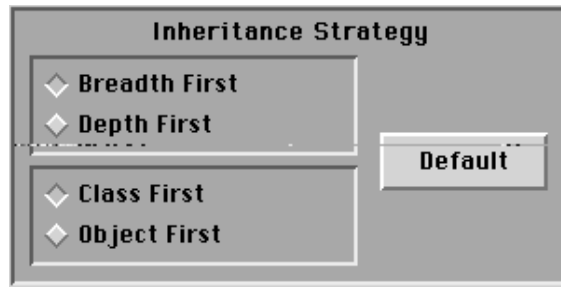
In both breadth-first and depth-first search, the order in which classes or objects are examined at each ply of the search tree is determined by their individual inheritance priorities or inheritance slots. In addition, the search may be constrained by the global inheritability settings in effect or by the local inheritability attributes of a given slot.

Private and public slots observe the same inheritance strategies. The private slot attribute controls the accessibility of the slot value and has nothing to do with inheritance.

Default

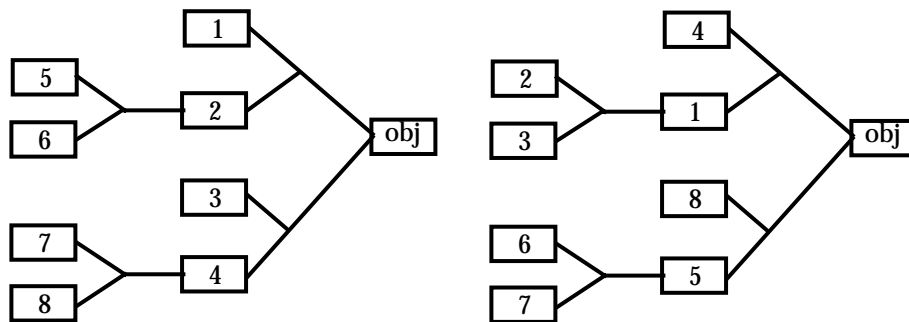
The system's default inheritance strategy is class-first and breadth-first. The default settings can be modified interactively through the Strategy Monitor window (from the Expert menu), during the course of evaluating a rule or method through the `Strategy` operator, or at the level of the individual

slot. In both the global Strategy Monitor window and the Meta-Slot editor, the inheritance strategy is controlled by the following radio buttons:



Global Control

The global inheritance strategy in effect for the entire system can be set either with the Strategy Monitor window (from the Expert menu) or via the Strategy operator in a rule or method, using the options @INHBREADTH and @INHPARENT. Clicking on the class-first or the object-first checkbox sets the inheritance strategy as follows. The global inheritance strategy in effect for the entire system can be set either with the Strategy Monitor window (from the Expert menu) or via the Strategy operator in a rule or method, using the options @INHBREADTH and @INHPARENT. Clicking on the class-first or the object-first checkbox sets the inheritance strategy as follows, where the diagram on the left represents breadth-first and the diagram on the right represents depth-first:



Breath-FirstDepth-First

Local Control

The global inheritance strategy can be overridden in the case of individual slots through the Meta-Slot editor.

Related Topics

- Objects
- Classes
- Properties
- Rules
- Methods
- Slot
- Meta-Slots

- Strategy
- Inheritability Strategy
- Inheritance
- Inheritance Priority Attribute
- Inheritance Slot Attribute
- Strategy Operator

InhMethod Operator

Definition

The *InhMethod operator* is used in the conditions or actions of methods to specify downward inheritance of the corresponding method from an including class, superclass, or parent object. Method inheritability allows an entire class of objects to share a single method, which is defined once for the class and automatically inherited by all instances.

Operand

The `InhMethod` operator takes one operand, which can be either the special reserved word `DEFAULT` or an explicitly named parent object from which to inherit the corresponding method.

```
InhMethod    DEFAULT
InhMethod    ParentObjectName
```

The operand can be an interpretation of the type `\slot_name\` that resolves to the desired slot `ParentObjectName`.

Effect

Execution of the method in which the `InhMethod` operator appears is suspended and an inherited method of the same name is executed. The method to be executed is sought by downward inheritance only (from class to instance, class to subclass, or parent object to component), subject to the global and local inheritance and inheritability strategies currently in effect. Methods can never be inherited upward. Once the inherited method finishes executing, the execution of the original, calling method resumes.

This operator also allows the developer to resolve inheritance conflicts by explicitly naming a parent object in the `InhMethod` operand. If no method can be triggered from the named parent object, the search for a corresponding method begins on the branch to which the object belongs. When the operand is `DEFAULT` and no parent object is explicitly named, inheritance conflicts are resolved based on the alphabetic order of the parent object names or inheritance priorities in the case of slots. However, if the order is important, it is recommended that you specify the method evaluation, rather than rely on the default behavior.

Result

When the `InhMethod` operator is used in a condition on the left-hand side of a method, the result produced by the operator is `TRUE` if the method is inherited, `FALSE` if a corresponding method does not exist or the parent object named through the operand has been deleted during the course of the session.

Example

Let's assume the following actions appear in a method attached to a subclass `Triangles` that belongs to a class `Figures`. The method is defined as a public one (inheritance enabled) and has the name `Init`:

```
InhMethod    Figures
Assign       SELF.width          SELF.height
```

The first action in this method demonstrates the use of the `InhMethod` operator to force the evaluation of another method of the same name before assigning the values. Let's assume it triggers inheritance from the class `Figures` of a public method (also named `Init`) with the following actions list:

```
Assign      SELF.originx      SELF.originx
Assign      SELF.originy     SELF.originy
```

Because the action in the first method triggers the method of the same name at the class level (`Figures`), the subclass `Triangles` inherits the new method down from its parent class before completing its own method actions list. In this case, the class `Figures` and the subclass `Triangles` share the same list of properties: `originx`, `originy`, `width`, and `height` and the definition of the method `Init` at the parent class avoids duplication of the initialization actions for its subclasses (or objects) whose properties it shares.

Related Topics

Objects	If Change Method
Classes	Inheritance
Conditions	Inheritance Strategy
Actions	Inheritability Strategy
Methods	Inheritance
Order of Sources Method	Inheritance Slot Attribute

InhValueDown Operator

Definition

The *InhValueDown operator* is used in the right-hand side actions of an Order of Sources method to specify downward inheritance of a public or private slot's value from that of a parent class or object.

Operand

The `InhValueDown` operator is valid only in the `THEN` actions list on the right-hand side of an Order of Sources. The `InhValueDown` operator takes one operand, which must be the special reserved word `DEFAULT`.

Effect

The value of the slot to which this Order of Sources method belongs is sought by downward inheritance (from class to instance, class to subclass, or parent object to component), subject to the global and local inheritance and inheritability strategies currently in effect.

Private and public slots both may obtain a value by downward inheritance. The private slot attribute controls the accessibility of the slot value and has nothing to do with inheritance.

Example

The following is the only valid form for an action using the `InhValueDown` operator:

```
InhValueDown DEFAULT
```

Related Topics

Objects	Inheritance
Classes	Inheritance Strategy
Properties	Inheritability Strategy
Slots	Inheritance Priority Attribute
Action	Inheritance Slot Attribute
Methods	InhValueUp Operator
Order of Sources Method	

InhValueUp Operator

Definition

The *InhValueUp operator* is used in the right-hand side actions of an Order of Sources method to specify upward inheritance of a public or private slot's value from that of an instance, subclass, or component (subobject).

Operand

The `InhValueUp` operator is valid only in the THEN actions list on the right-hand side of an Order of Sources. The `InhValueUp` operator takes one operand, which must be the special reserved word `DEFAULT`.

Effect

The value of the slot to which this Order of Sources method belongs is sought by upward inheritance (from instance to class, subclass to class, or component to parent object), subject to the global and local inheritance and inheritability strategies currently in effect.

Private and public slots both may obtain a value by upward inheritance. The private slot attribute controls the accessibility of the slot value and has nothing to do with inheritance.

Example

The following is the only valid form for an action using the `InhValueUp` operator:

```
InhValueUp DEFAULT
```

Related Topics

Object	Inheritance
Classes	Inheritance Strategy
Properties	Inheritability Strategy
Slots	Inheritance Priority Attribute
Actions	Inheritance Slot Attribute
Methods	InhValueDown Operator
Order of Sources Method	

Init Value Attribute

Definition

An *Init Value Attribute* can be used to declare an initialization value for individual public and private slots.

Effect

A slot that has an initial value declared will automatically be initialized to that value either when the knowledge base file containing the initial value declaration is loaded or when the state of the system is reinitialized with the Restart Session command. If the inheritability strategy of the initialized slot permits, the system automatically propagates the value to the children slots according to the inheritability strategy defined for the initialized slot. Whether the slot is public or private has no effect on slot value initialization.

Notice the difference between an initial value and the assignment made through the `RunTimeValue` operator. The initial value specifies a value to be set and propagated at system initialization time; `RunTimeValue` specifies a default value to be set dynamically during inference processing when processed in the Order of Sources method. Also, no `If Change` method is triggered when a slot's value is determined by an initial value, whereas `RunTimeValue` will trigger the corresponding `If Change` method.

Creation

The initial value is specified or edited by typing into the box labeled `Init Value Public` or `Init Value Private` in the Meta-Slot editor. The supplied value can be a string, integer, or boolean value (including the keyword `NOTKNOWN`). String values must appear between double quotes ("a_string").

If you want to specify an initial value for a slot that is different from its parent's initial value declaration, you can modify the meta-slot attribute local to the slot. Initial values that are declared locally override any potentially inheritable initial value declarations.

Inheritance

The value of the slot can be made uninheritable by typing the value into the `Private` box, otherwise type the value in the `Public` box. The inheritability of a slot's initial value when declared overrides either local or global inheritability strategies currently in effect. An initial value declared locally also overrides any potentially inheritable initial value declarations.

Related Topics

Boolean Value
Objects
Properties
Slots
Data Types

Methods
Order of Sources Method
`RunTimeValue` Operator
Inheritability Strategy
Meta-Slots

INT2STR Function

Definition

The *INT2STR function* is used in expressions to convert an integer value to an equivalent character string. The expression can appear on the left-hand side or right-hand side of rules and methods.

Syntax

The function consists of the word `INT2STR` followed by one or two arguments in parentheses:

```
INT2STR(n)  
INT2STR(n, f)
```

Argument

Each argument may be any expression yielding a result of the appropriate type:

- The first argument (*n*) is the integer to be converted.
- The optional second argument (*f*) is a string specifying the format under which the first argument is to be converted. See “Integer Formats” for the syntax and meaning of this string.

Argument *n* may also yield a floating point value, which will be truncated to the next lower integer (toward zero) before being converted. The argument expressions may include patterns or interpretations.

Result

The function returns a string result representing the numeric value of argument *n*, converted according to format *f*. If no format argument is given, the default system format for integers (defined in the `ckbres.format` module in the file `nrxrun.dat`) is used.

Examples

The following examples illustrate the results of the `INT2STR` function:

```
INT2STR(98)      = "98"  
INT2STR(98.6)   = "98"  
INT2STR(-98.6)  = "-98"  
INT2STR(79, "x") = "4f"
```

Related Topics

Expressions	Integer Formats
String Constants	Patterns
Integer Constants	Interpretations
Floating Point Constants	STR2INT Function

Integer Constants

Definition

An *integer constant* is a sequence of characters that stand directly for an integer (whole number) value.

Syntax

An integer constant consists of one or more decimal digits (0–9), optionally preceded by a sign (+ or -). It must not include embedded spaces, commas, a decimal point, or an exponent.

Examples

The following are valid integer constants:

6
-27
+441
0
16777216

The following are not:

abc	Not a number
6+5	Expression, not a constant
23a	Contains an invalid character
16 777 216	Contains embedded spaces
16,777,216	Contains embedded commas
98.6	Contains a decimal point
125e3	Contains an exponent

Related Topics

Data Types

Integer Formats

Floating Point Constants

Expressions

Integer Formats

Definition

An *integer format* specifies the representation of an integer value in text form for input and output purposes.

Syntax

This section defines the syntax of format elements for integer-valued properties only. See the section titled “Formats” for the syntax of formats in general.

The following special characters are meaningful in integer formats:

d	Decimal representation
X	Hexadecimal representation with capital letters A–F for digit values 10–15
x	Hexadecimal representation with lowercase letters a–f for digit values 10–15
0	significant digits only

Any of these may optionally be followed by a series of zeros (0) defining the minimum number of digits to be used in representing the number. For example, the format `d000` denotes a decimal number at least three digits long.

Like all formats, those for integers may include strings of literal characters enclosed in double quotation marks (" . . . "), and may also include the wild-card character (*). Format elements beginning with an exclamation point (!) are ignored in database transactions; they are meaningful only for direct interaction with the user via the screen and keyboard.

Input

On input, each element in the format list is tried in order until one of them matches the input text. If no match is found, the input is rejected and an error message is displayed on the screen. The following conventions apply:

- Odd-numbered elements in the format list (the first, third, and so on) produce a positive result, even-numbered elements (the second, fourth, and so on) produce a negative result.
- Input values of any length are recognized; zeros (0) used in the format to specify the number of digits in the data value are ignored.
- In hexadecimal representation, no distinction is made between uppercase digits A–F and lowercase a–f. Both forms are recognized, and may even be mixed in the same number; the case explicitly specified by the format itself (X or x) is ignored.
- Strings of literal characters enclosed in double quotation marks must match exactly, except that no distinction is made between upper- and lowercase letters.
- The wild-card character (*) matches any sequence of zero or more characters.

Output

On output, only the first one or two elements in the format list are used:

- The first format element is used for positive and zero values, the second for negative values; any further elements in the list are ignored. If there is no second element, the first is used for all output values.
- A series of zeros (0) within a format element specifies the minimum number of digits to be used in the output representation. Numbers shorter than this will be padded with leading zeros; longer numbers will be represented in full, using more than the specified number of digits.

- Strings of literal characters enclosed in double quotation marks are reproduced exactly in the output.
- The wild-card character (*) is ignored on output.

Default

The default system format for integers is defined in the `ckbres.format` module in the file `nrxrun.dat`. The standard default format is

`d`

denoting decimal representation in the minimum required number of digits.

Examples

The following examples illustrate the use of integer formats:

Format: `d000;"+d000;"-d000`

Value	Output	Comments
23	023	Leading zero to fill
1234	1234	Exceeds specified length
-23	-023	No second element; uses first

Input	Value	Comments
23	23	Matches first element
-23	23	Matches first element
+23	23	Matches third element
23.0	NOTKNOWN	No match; use <code>d000*</code>

Format: `"0x"X0000;;d`

Value	Output	Comments
254	0x00FE	Leading zeros to fill
-1	0xFFFFFFFF	Exceeds specified length

Input	Value	Comments
0xfe	254	Case is irrelevant
254	254	Matches third element

Format: `d000*;"minus"d000*`

Value	Output	Comments
23	023	Leading zero to fill
1234	1234	Exceeds specified length
-23	minus 023	Negative uses second element

Input	Value	Comments
23	23	Matches first element
-23	-23	Matches first element
minus 23	-23	Matches second element
plus 23	NOTKNOWN	No match
23.7	23	No rounding; wild card discards fractional part

In the last example, notice that both input values (`0xfe` and `254`) will be displayed on output as `0x00FE`.

Related Topics

Formats
Format Attribute

Integer Constants
Floating Point Formats

Interpretations

Definition

An *interpretation* is used in an expression to refer to an object, class, or property indirectly, via the value of a slot calculated at runtime.

Syntax

Typically, an interpretation can be used wherever an object, class, or property name would be valid in an expression, although it is specifically not allowed in the `SendMessage` operator expression. It consists of the name of a slot enclosed between backslashes (`\ . . . \`). It may optionally be preceded by a string of characters, called the root string, enclosed in single quotation marks (`' . . . '`). The root string or the variable name (but not both) may be empty.

If the slot used in the interpretation is a private slot, the interpretation can only appear in the method attached to the slot and the `SELF` keyword must be used to refer to the private slot name. Interpretations that appear in rule conditions and actions must be made on public slots.

Meaning

The slot named within the backslashes is evaluated and the resulting string is substituted in its place in the expression. If the interpretation includes a root string, it is concatenated together with the value of the slot to form the required object, class, or property name.

If the slot named between the backslashes is not of type `STRING`, its value is converted into an equivalent string of characters before being used. In particular, floating point values are truncated to their integer part only, since the decimal point (`.`) is not a valid character in an object, class, or property name.

An interpretation may be embedded within a pattern, but a pattern may not be embedded within an interpretation.

Examples

The following are valid class, object, or property interpretations:

```
\which_client\  
\whic__client.name\  
' '\which_client\  
'tank_'\n\  
'tank_'\tank.number\  
<|\component_class\|>  
{\warehouse.inventory\  
regular_tank_1.\tank.level\  
auxiliary_tank_1.'aux_'\tank.level\  
'regular_tank_'\tank.number\.\tank.level\  
\which_company.name\.\which_client\  

```

The following are not valid interpretations:

<code>\which_client</code>	Backslashes not balanced.
<code>tank_\n\</code>	No quotes around root.
<code>'_tank'\n\</code>	Invalid form for identifier.
<code>'tank_'\m+n\</code>	Expression inside backslashes.
<code>'part_'\<Part>.number\</code>	Pattern inside backslashes.

If the value of `empty_tank` is the string `tank_3`, then the expressions

`\empty_tank\.capacity`

and

`''\empty_tank\.capacity`

are both equivalent to

`tank_3.capacity`

Similarly, if both `n` and `tank.number` are equal to 3, then

`'tank_'\n\.capacity`

and

`'tank_'\tank.number\.capacity`

are again equivalent to

`tank_3.capacity`

If the value of `component_class` is the string `Switch`, then the existential pattern

`<|\component_class\|>`

refers to all existing instances of class `Switch`. If the value of `warehouse.inventory` is `parts_in_stock`, then the universal pattern `{\warehouse.inventory\}`

denotes all components (subobjects) of the object `parts_in_stock`.

Related Topics

Objects

Classes

Properties

Slots

Data Types

Identifiers

Expressions

Patterns

Interrupt Operator

Definition

The *Interrupt operator* is used in the conditions or actions of methods to interrupt the execution of the method and return control of the system to the user.

Operand

The `Interrupt` operator takes one operand, which must be the boolean constant `TRUE`. The following is the only valid form for an action using the `Interrupt` operator:

```
Interrupt    TRUE
```

Effect

Execution of the method containing the `Interrupt` operator is interrupted, displaying an alert box with a message. For example, for an `Order of Sources` method:

```
Interrupt in Sources slot of Flap.doodle.
```

or, for an `If Change` method:

```
Interrupt in Action slot of Flap.doodle.
```

During the interruption, the user is free to activate other windows, edit the knowledge base, invoke commands, or take any other desired action. Clicking the `Continue` button in the session control panel of the `Rules Element` main window resumes execution of the suspended method from the point of the interruption.

Result

When the `Interrupt` operator is used in a condition on the left-hand side of a method, the result produced by the operator is always `TRUE`.

Related Topics

Properties	Order of Sources Method
Actions	If Change Method
Methods	

LENGTH Function

Definition

The *LENGTH function* is used in expressions to find the number of objects matching a given pattern. The expression can appear on the left-hand side or right-hand side of rules and methods.

Syntax

The function consists of the word `LENGTH` followed by a single argument in parentheses:

```
LENGTH(p)
```

Argument

The argument may be any existential pattern with a property name specified. Universal patterns are not allowed.

Note: The pattern must include a property name or unexpected side-effects in gating may result. If desired, you can execute your own C routine to get the number of objects attached to a class.

Result

The function returns an integer result equal to the number of objects in the list corresponding to the given pattern.

Examples

The following examples illustrate the results of the `LENGTH` function. If class `Client` has 22 instances, object `job_queue` has 12 components, and object `orders_pending` has none, then

```
LENGTH(<Client>.name)      = 22
LENGTH(<job_queue>.value)  = 2
LENGTH(<orders_pending>)  = 0
```

The following expressions are invalid:

```
LENGTH(<Client>)          = 22
LENGTH(Client)            Not a pattern
LENGTH(Client.name)      Not a pattern
LENGTH({Client})         Universal patterns not allowed
```

Related Topics

Expressions

Objects

Classes

Patterns

Integer Constants

LN Function

Definition

The *LN function* is used in expressions to find the natural (Napierian) logarithm of a floating point number. The expression can appear on the left-hand side or right-hand side of rules and methods.

Syntax

The function consists of the word `LN` followed by a single argument in parentheses:

```
LN(x)
```

Argument

The argument may be any expression yielding a numerical result greater than 0.0. The expression may include patterns or interpretations.

If the value of the argument expression is an integer, it will be converted to an equivalent floating point value.

Result

The function returns a floating point result equal to the logarithm of the argument to the Napierian base e ($= 2.71828$).

If the argument expression does not produce a numerical value, an error message is posted and the function result is `NOTKNOWN`.

Examples

The following examples illustrate the results of the LN function:

```
LN(0.0001)          = -9.21
LN(1 / 2.71828)     = -1.0
LN(SQRT(2.71828))  = 0.5
LN(2.71828)        = 1.0
LN(10000)          = 9.21
```

Related Topics

Expressions	Interpretations
Floating Point Constants	LOG Function
Integer Constants	EXP Function
Patterns	

LoadKB Operator

Definition

The *LoadKB operator* is used in the conditions or actions of a rule or method to load or enable a knowledge base.

Operands

The `LoadKB` operator takes one or two operands:

- The first operand is a string constant or an interpretation which evaluates to a string constant (using the `@v(object.prop)` syntax) specifying the name of the file containing the knowledge base to be loaded. It must be between double quotes.
- The optional second operand specifies the knowledge base's load level, and must be one of the following:

```
@LEVEL=ENABLE ;
@LEVEL=DISABLEWEAK ;
@LEVEL=DISABLESTRONG ;
```

(Note that the closing semicolon is required.) If the second operand is omitted, a load level of `ENABLE` is assumed by default.

LoadKB Dialog

When entering a LoadKB action in the Rule editor or Method editor, clicking in the space for the second operand displays a special dialog box for specifying the load level interactively, rather than by explicitly typing in the keywords listed above:



Effect

The knowledge base named as the first operand is loaded into memory from a file and given the load level specified by the second operand. Definitions loaded from the knowledge base are added to those already present in memory. If the designated knowledge base is already loaded, its load level is simply changed to that specified by the second operand.

Load Levels

The effects of the various load levels are as follows:

- ENABLE: All definitions in the knowledge base are fully effective and operational, including objects, classes, properties, rules, and methods.
- DISABLEWEAK: Object, class, and property definitions in the knowledge base are in effect. Rules and methods are defined, but are temporarily disabled and unavailable for inference processing; they can later be reenabled by specifying load level `ENABLE`. Any such disabled rules or methods already on the agenda remain there and will be processed normally.
- DISABLESTRONG: Object, class, and property definitions in the knowledge base are in effect. Rules and methods are defined, but are temporarily disabled and unavailable for inference processing; they can later be reenabled by specifying load level `ENABLE`. Any such disabled rules or methods already on the agenda are removed from the agenda and will not be processed.

Examples

The following are examples of actions using the LoadKB operator:

```
LoadKB      "Inventory.tkb"
LoadKB      "Inventory.ckb"      @LEVEL=ENABLE
LoadKB      "Inventory.ckb"      @LEVEL=DISABLEWEAK
LoadKB      "@v(object.prop) "    @LEVEL=DISABLESTRONG
```

Related Topics

Rules

Methods

Actions

Objects

Classes

Properties

Agenda

String Constants

UnloadKB Operator

LOG Function

Definition

The *LOG function* is used in expressions to find the common (decimal) logarithm of a floating point number. The expression can appear on the left-hand side or right-hand side of rules and methods.

Syntax

The function consists of the word LOG followed by a single argument in parentheses:

```
LOG(x)
```

Argument

The argument may be any expression yielding a numerical result greater than 0.0. The expression may include patterns or interpretations.

If the value of the argument expression is an integer, it will be converted to an equivalent floating point value.

Result

The function returns a floating point result equal to the logarithm of the argument to the base 10.

If the argument expression does not produce a numerical value, an error message is posted and the function result is NOTKNOWN.

Examples

The following examples illustrate the results of the LOG function:

```
LOG(0.0001)   = -4.0
LOG(0.1)      = -1.0
LOG(SQRT(10)) = 0.5
LOG(10)       = 1.0
LOG(10000)    = 4.0
```

Related Topics

Expressions	Patterns
Floating Point Constants	Interpretations
Integer Constants	LN Function

MAX Function

Definition

The *MAX function* is used in expressions to find the largest of a set of values. The expression can appear on the left-hand side or right-hand side of rules and methods.

Syntax

The function consists of the word `MAX` followed by any number of arguments in parentheses:

```
MAX(x1, x2, . . . , xn)
```

Arguments

Each argument may be any arbitrary expression. The expressions may include existential patterns or interpretations; universal patterns are not allowed.

Argument values may be of any type, but the types must be comparable (either all the same or all numeric). If some are integers and some floating point, the integers will be converted to equivalent floating point values before comparison.

Result

The function returns the largest of the argument values it receives. For arguments that include patterns, it finds the largest value in the corresponding list.

Integers and floating point values are compared numerically, strings lexically, and dates and times chronologically. In string comparisons, equivalent uppercase and lowercase letters (such as `A` and `a`) are considered identical. In boolean comparisons, `TRUE` is considered greater than `FALSE`.

If the argument values are not of comparable types, the function result is `NOTKNOWN`.

Examples

The following examples illustrate the results of the `MAX` function:

```
MAX(365,240,577)           = 577
MAX(98.6,37.0,-273.18)    = 98.6
MAX(12,12.0)              = 12.0
MAX(12,12.3)              = 12.3
MAX(12,11.7)              = 12.0
MAX("Hickory", "Dickory", "Dock") = "Hickory"
MAX("boo", "ooojum")      = "ooojum"
MAX("ABC", "xyz")         = "xyz"
MAX("abc", "XYZ")         = "XYZ"
MAX("", "SHAZAM!")        = "SHAZAM!"
```

```

MAX( DATE(1776,7,4), DATE(1789,7,14) ) = DATE(1789,7,14)
MAX( TIME(8,4,23), TIME(3,6,11) )      = TIME(8,4,23)
MAX( TRUE, FALSE ) = TRUE
MAX(123, "456") = NOTKNOWN

```

If class `Tank` has four instances with capacity values of 6.3, 14.5, 12.9, and 9.0, then

```
MAX(<Tank>.capacity) = 14.5
```

Related Topics

Expressions
Data Types
Patterns

Interpretations
MIN Function

Member Operator

Definition

The *Member operator* is used in the conditions of a rule or method to test whether an object belongs to a given class or list.

Operands

The `Member` operator takes two operands:

- The first operand is either a single object or a list of objects specified by a pattern.
- The second operand is a list of objects specified by a pattern. This operand must use the pattern matching syntax.

The second operand is commonly a list of objects satisfying some qualification or relation, as determined by a prior condition within the same rule or method.

Result

The result produced by the `Member` operator is `TRUE` if the first operand is a member of the class or list designated by the second, `FALSE` if it isn't. If the first operand is a pattern, the condition tests whether at least one of the objects in the corresponding list (for an existential pattern) or all of them (for a universal pattern) also belong to the second class or list. The contents of the first list are then reduced to the intersection of the two.

Examples

The following are examples of conditions using the `Member` operator:

```

Member    the_stock    <Portfolio>
Member    <Portfolio>   <Common_Stock>
Member    {Portfolio}  <Common_Stock>

```

Related Topics

Rules
Methods
Conditions

Objects
Patterns
NotMember Operator

Meta-Slots

Definition

Meta-slots are attributes associated with a slot (a property associated with a class or object), governing its inheritability and relationships with the user interface.

Variations

The following meta-slots can be associated with an individual slot:

- The *public/private* option controls whether the slot value will be accessible by rules and methods (public slot) or by methods only (private slot).
- The *inheritance strategy* controls the inheritance of the slot's value from including classes and parent objects.
- The *inheritability strategy* controls the inheritance of the slot and its value by subclasses, instances, and components.
- The *inheritance priority* defines the priority with which the slot or its value can be inherited.
- The *inheritance slot* allows the inheritance priority to be determined dynamically at run time, rather than fixed unalterably in advance.
- The *inference priority* defines the slot's priority and behavior during inference processing.
- The *inference slot* allows the inference priority to be determined dynamically at run time, rather than fixed unalterably in advance.
- The *format attribute* defines the way in which the slot's value is displayed on the screen.
- The *prompt line attribute* defines the text to be displayed on the screen when requesting the slot's value from the user. This meta-slot can be inherited downward.
- The *why attribute* allows you to customize the Why information for a particular slot.
- The *comment attribute* helps document the slot's meaning or usage for the benefit of the application developer.
- The *init value field* specifies an initialization value for the individual slot to be used when the knowledge base is loaded. The inheritability strategy of this meta-slot is specified for each value.
- The *question window attribute* lets you associate the component of your application interface that the system will use to solicit the slot's value from the end-user.
- The *data validation attribute* lets you predetermine the range of input or list of strings that the system will accept from the end-user when the value of the slot is sought. This meta-slot can be inherited downward.

Creation

Meta-slots are specified by editing the contents of the relevant boxes in the Meta-Slot editor.

Indication

The presence of one or more meta-slot definitions for an individual slot is indicated by a solid-colored box at the right end of the property's value in the Class or Object editor. If no meta-slots are defined, the box is displayed in outline only. Clicking on the box with the mouse brings up the Meta-Slot editor, allowing the meta-slots to be defined or modified.

Related Topics

Objects	Format Attribute
Classes	Question Window
Properties	Data Validation
Slots	Init Value Attribute
Inheritance Strategy	Why Attribute
Inheritability Strategy	Comment Attribute
Inheritance Priority Attribute	Inference Priority Attribute
Inheritance Slot Attribute	Inference Slot Attribute
Prompt Line Attribute	

Methods

Definition

A *method* is an attribute attached to an object, class, property, public slot or private slot, consisting of a sequence of actions to be executed under certain conditions during inference processing. There are two general categories of methods. *User-defined* methods that may be triggered through the use of the `SendMessage` operator during the course of evaluating rules and other methods. *System* methods are automatically triggered by the inference engine under predefined circumstances. Unlike public slots, private slots must have their attached method triggered explicitly by a `SendMessage` operator.

Structure

The method consists of most importantly a sequential list of actions, similar to those on the right-hand side of a rule. If desired, the method can be structured exactly like a rule including a list of conditions and two separate consequent lists of actions. Unlike rules, methods have no hypothesis component. Methods can also accept local arguments which you use in the method actions and conditions. Generic methods can use the `SELF` variable to represent the current class or object.

Creation

Creation begins by typing the name of the method in the `Method` field of the Method editor. Or you can display the local popup menu for the edit line and choose the Select Method option to make a selection from the list of existing methods. System methods are usually attached at the level of the individual slot (optionally to a class or object, see the Order of Sources Method and If Change Method topics for further details). User-defined methods can be attached to a property, a class, or an object, as well as a

public or private slot. The atom name to which the method is attached is specified in the `Attach To` field.

If local arguments will be passed to the method by the `SendMessage` operator, the method itself defines the characteristics of the arguments locally. The `Local Arguments` component of the Method editor lets you specify the argument name for use in the method's conditions and actions. The name you specify must be preceded by an underscore (`_`). Other fields determine the local argument's usage for that particular method.

Invocation

User-defined methods are not limited to slots, but must be explicitly triggered through a `SendMessage` operator that appears in a condition or action of a rule or method. The application developer has the choice to send the message at startup or from the interface using either the scripting language or using the Rules Element application programming interface. Whenever a method is triggered by the `SendMessage` operator, the system executes the complete list of actions.

There are two types of system methods that are available at the level of the individual public slot:

- The *order of sources method* is triggered automatically when the value of a public slot is needed in the course of inference processing and was found to be `UNKNOWN`.

Note: In the case of a private slot an Order of Sources method can be attached, but the system is unable to trigger the method automatically. The application developer is required to use the `SendMessage` operator to explicitly trigger the system method of a private slot.

- The *if change method* is triggered automatically when the value of a public or private slot is changed in the course of inference processing.

The list of conditions is optional for all methods. If no conditions are present, the system automatically executes the Then actions list when the method itself is triggered. If method conditions are present, the system executes one of two different lists of consequent actions (Then or Else) depending on whether the method is satisfied or not.

For the method to be satisfied, all of its conditions must evaluate to `TRUE`. The conditions are thus implicitly linked by the logical "and" operator. To achieve the effect of a logical "or," use the boolean `OR` operator within a single condition.

If present, conditions within a method are always evaluated sequentially, in the order they appear in the method definition; unlike rule conditions this evaluation order is not altered by the inference priorities of the data involved.

If the system tries to trigger a method for a property name, it first tries the slot to which the property belongs (`object.prop` or `class.prop`). When no slot has been defined, the system will try the property definition itself.

If no method is specified at the level of the addressee (in the case of a user-defined method) or at the level of the slot (in the case of a system method), a substitute method of the same name will be sought by downward inheritance. See the section on "Inheritance" for more details.

Strategy

Execution of system methods that are under the control of the inference engine (If Change and Order of Sources) is normally enabled by default, but can be disabled if necessary by changing the global inference strategy. This can be done in either of two ways:

- Interactively through the Strategy Monitor window (from the Expert menu), by turning off the `If Change Actions` option or the `Order of Sources Actions` option.
- Dynamically in the course of inference processing itself, via the `Strategy` operator in a condition or action of a rule or method.

Note: The `SendMessage` operator can be used to explicitly trigger any method. The method triggered by the `SendMessage` operator is not affected by any of the strategy settings and will actually be treated as a user-defined method by the inference engine.

Forward Chaining

Actions that appear in the conditions list or actions list of a method may forward-chain data from public slots to relevant rules depending on the inferencing strategies currently in effect. The method actions include: `Assign`, `Retrieve` (from a database), and `Execute` (using an external routine). From the method conditions list only the results of the `Retrieve` and `Execute` actions may be forward-chained. The `Assign` operator has no effect on forward chaining from the conditions list. See the individual operator topics for details.

Data that belongs to a private slot that appears in a method condition or action cannot trigger forward chaining since private slot data cannot appear in the conditions or actions of rules. Only data that belongs to public slots can trigger forward chaining.

Methods are not affected by the results of actions or gates because they do not have hypotheses to be considered for evaluation.

Inheritance

Methods can only be inherited downward (from a class to its instances or subclasses, or from an object to its components), never upward. The search through the parent tree hierarchy is directed by the global inheritance strategy and can be class or object-first and depth or breadth-first. If the method should not be inherited, change the `Public` option to `Private` in the Method editor.

When an inheritance conflict exists between two parent objects or classes at the same level, the application developer can use the `InhMethod` operator to override the default inheritance strategy by specifying the parent object upon which to begin the search. When the inheritance conflict occurs between two slots at the same level, the application developer can set the inheritance priority of the slots to override the default inheritance strategy. If neither approach is used, by default the system chooses the method attached to the parent whose name appears first in alphabetic order.

Private and public slots observe the same inheritance strategies. The private slot attribute controls the accessibility of the slot value and has nothing to do with inheritance.

Example

Let's assume the following actions appear in two methods attached to the subclasses `Triangles` and `Rectangles` that belong to a class `Figures`. The method attached to `Triangles` is defined as a public one (inheritance enabled) and has the name `ComputeArea`:

```
Assign      (SELF.width*SELF.height)/2      SELF.area
```

The second method attached to `Rectangles` is also defined as a public one and has the same name `ComputeArea`:

```
Assign      SELF.width*SELF.height          SELF.area
```

To trigger these methods, let's assume we have a rule with the following `SendMessage` action:

```
SendMessage      "ComputeArea"      @TO:<Figures>
```

Because the `SendMessage` operator in this rule specifies a pattern match on the class `Figures` as its addressee, the message is received by each object that belongs to the class. Let's assume the following objects exist: `Rect_1`, `Rect_2`, `Tri_1`, and `Tri_2` and that no method is attached at their level. In case, each object will automatically inherit the method `ComputeArea` defined at the level of its parent class and the specific values for the properties `width` and `height` may be supplied by the objects themselves or may be obtained from a question or some other means.

In this example, definition of the method `ComputeArea` at the level of the parent classes (`Triangles` and `Rectangles`) avoids duplication of the area computation action for each object whose properties they share.

Related Topics

Objects	Inheritance
Classes	Inheritance Strategy
Properties	Inheritance Priority
Conditions	Strategy
Actions	If Change Method
Rules	Order of Sources Method
Slots	InhMethod Operator
Inference	SendMessage Operator
Forward Chaining	

MIN Function

Definition

The *MIN function* is used in expressions to find the smallest of a set of values. The expression can appear on the left-hand side or right-hand side of rules and methods.

Syntax

The function consists of the word `MIN` followed by any number of arguments in parentheses:

```
MIN(x1, x2, . . . , xn)
```

Arguments

Each argument may be any arbitrary expression. The expressions may include existential patterns or interpretations; universal patterns are not allowed.

Argument values may be of any type, but the types must be comparable (either all the same or all numeric). If some are integers and some floating point, the integers will be converted to equivalent floating point values before comparison.

Result

The function returns the smallest of the argument values it receives. For arguments that include patterns, it finds the smallest value in the corresponding list.

Integers and floating point values are compared numerically, strings lexically, and dates and times chronologically. In string comparisons, equivalent uppercase and lowercase letters (such as A and a) are considered identical. In boolean comparisons, TRUE is considered greater than FALSE.

If the argument values are not of comparable types, the function result is NOTKNOWN.

Examples

The following examples illustrate the results of the MIN function:

```

MIN(365,240,577)           = 240
MIN(98.6,37.0,-273.18)    = -273.18
MIN(12,12.0)              = 12.0
MIN(12,12.3)              = 12.0
MIN(12,11.7)              = 11.7
MIN("Hickory","Dickory","Dock") = "Dickory"
MIN("boo","boojum")       = "boo"
MIN("ABC","xyz")          = "ABC"
MIN("abc","XYZ")          = "abc"
MIN("", "SHAZAM!")        = ""
MIN( DATE(1776,7,4), DATE(1789,7,14) ) = DATE(1776,7,4)
MIN( TIME(8,4,23), TIME(3,6,11) )      = TIME(3,6,11)
MIN(TRUE,FALSE)            = FALSE
MIN(123,"456")             = NOTKNOWN

```

If class Tank has four instances with capacity values of 6.3, 14.5, 12.9, and 9.0, then

```
MIN(<Tank>.capacity) = 6.3
```

Related Topics

Expressions

Data Types

Patterns

Interpretations

MAX Function

MINUTE Function

Definition

The *MINUTE function* is used in expressions to extract the minute field of a date or time. The expression can appear on the left-hand side or right-hand side of rules and methods.

Syntax

The function consists of the word `MINUTE` followed by a single argument in parentheses:

```
MINUTE ( d )
```

Argument

The argument may be any expression yielding a date or time result. The expression may include patterns or interpretations.

Result

The function returns an integer result equal to the minute field of the argument. For date arguments, the result ranges from 0 to 59.

If the argument expression does not produce a date or time value, an error message is posted and the function result is `NOTKNOWN`.

Examples

The following examples illustrate the results of the `MINUTE` function:

```
MINUTE ( DATE ( 1492 , 10 , 12 ) )      =  0
MINUTE ( DATE ( 1981 , 6 , 8 , 21 , 8 , 46 ) ) =  8
MINUTE ( TIME ( 8 , 4 , 23 ) )         =  4
MINUTE ( TIME ( 3 , 6 , 11 , 22 , 34 , 17 ) ) = 34
MINUTE ( "October 12, 1492" )         = NOTKNOWN
```

Related Topics

Expressions

[DATE Function](#)

[TIME Function](#)

[Patterns](#)

[Interpretations](#)

[YEAR Function](#)

[MONTH Function](#)

[DAY Function](#)

[HOUR Function](#)

[SECOND Function](#)

[WEEKDAY Function](#)

[YEARDAY Function](#)

[NOW Function](#)

MOD Function

Definition

The *MOD function* is used in expressions to find the remainder of one floating point or integer number modulo of another. The expression can appear on the left-hand side or right-hand side of rules and methods.

Syntax

The function consists of the word `MOD` followed by two arguments in parentheses:

```
MOD(x, y)
```

Arguments

Each argument may be any expression yielding a numerical result. The expressions may include patterns or interpretations.

Result

The function returns a floating point result equal to the remainder of the first argument modulo the second ($x \bmod y$) if one or both arguments are floats. If both arguments are integers, the function will also return an integer. This value is defined as the difference between y and the next smaller whole multiple of x . Truncation is always toward zero, yielding a result of the same sign as x .

If either argument expression does not produce a numerical value, an error message is posted and the function result is `NOTKNOWN`.

Examples

The following examples illustrate the results of the `MOD` function:

```
MOD( 8, 3) = 2
MOD( 8, -3.0) = 2.0
MOD(-8, 3.0) = -2.0
MOD(-8, -3) = -2
MOD(8.5, 3.1) = 2.3 (= 8.5 - 2 * 3.1)
```

Related Topics

Expressions

Floating Point Constants

Integer Constants

Patterns

Interpretations

MONTH Function

Definition

The *MONTH function* is used in expressions to extract the month field of a date or time. The expression can appear on the left-hand side or right-hand side of rules and methods.

Syntax

The function consists of the word `MONTH` followed by a single argument in parentheses:

```
MONTH(d)
```

Argument

The argument may be any expression yielding a date or time result. The expression may include patterns or interpretations.

Result

The function returns an integer result equal to the month field of the argument. For date arguments, the result ranges from 1 (January) to 12 (December).

If the argument expression does not produce a date or time value, an error message is posted and the function result is NOTKNOWN.

Examples

The following examples illustrate the results of the MONTH function:

```
MONTH( DATE( 1492, 10, 12 ) )      = 10
MONTH( DATE( 1981, 6, 8, 21, 8, 46 ) ) = 6
MONTH( TIME( 8, 4, 23 ) )         = 0
MONTH( TIME( 3, 6, 11, 22, 34, 17 ) ) = 6
MONTH( "October 12, 1492" )      = NOTKNOWN
```

Related Topics

Expressions	HOUR Function
DATE Function	MINUTE Function
TIME Function	SECOND Function
Patterns	WEEKDAY Function
Interpretations	YEARDAY Function
YEAR Function	NOW Function
DAY Function	

Multi-Values

Definition

A *multi-value* is a string value representing a series of individual items that can be extracted or manipulated separately.

Syntax

A multi-value consists of one or more string-valued items separated by commas. (If there is just one item, the multi-value is indistinguishable from a simple string value representing the item.) Leading and trailing spaces around each item are ignored, but internal spaces within an item are significant.

The length of a multi-value string is limited by default to no more than 2,048 characters, but may be extended by setting NXP_BUFSIZE.

Operations

The following execute routines perform various operations on multi-values:

GetMultiValue	ComputeMultiValue
SetMultiValue	LinkMultiValue
TestMultiValue	AtomName

Example

The string

```
"London , Paris, New York , Tokyo "
```

is a legal multi-value consisting of the four items

```
London
Paris
New York
Tokyo
```

Notice that the spaces before and after each item are ignored, but the internal space in `New York` is significant.

Related Topics

String Constants	TestMultiValue Routine
Execute Routines	ComputeMultiValue Routine
GetMultiValue Routine	LinkMultiValue Routine
SetMultiValue Routine	AtomName Routine

Refer to Chapter Two, “Execute Library Routines” for a description of specific routines.

No Operator

Definition

The *No operator* is used in the conditions of a rule or method to test whether a boolean value or boolean expression is `FALSE`.

Operands

The `No` operator takes a single operand, which must be either a boolean-valued slot, a list of such slots specified by a pattern, or a boolean expression.

Result

The result produced by the `No` operator is the logical inverse of its boolean operand: `TRUE` if the operand is `FALSE`, `FALSE` if the operand is `TRUE`. If the operand includes a pattern, the condition tests whether the overall result of the pattern match is `FALSE`. Thus for an existential pattern, the result is `TRUE` if all values in the corresponding list are `FALSE`; for a universal pattern, it is `TRUE` if at least one value in the list is `FALSE`. If the operand is a boolean expression, the result is the logical inverse of the value of the resolved expression (either `TRUE` or `FALSE`).

Examples

The following are examples of conditions using the `No` operator:

```
No    credit_approved
No    switch_1.on
No    <Switch>.on
No    {Switch}.on
```

Related Topics

Rules	Boolean Constants
Methods	Patterns
Conditions	Yes Operator
Boolean Expressions	

NoInherit Operator

Definition

The *NoInherit operator* is used in the conditions or actions of methods to prevent inheritance of the standard default behavior for the given method.

Operand

The `NoInherit` operator takes one operand, which must be the boolean constant `TRUE`. The following is the only valid form for an action using the `NoInherit` operator:

```
NoInherit    TRUE
```

Effect

The `NoInherit` operator is meaningful only when used alone, as the only action in a method. The standard default behavior for the given method is disabled, preventing any inheritance of methods or values from other classes and objects. In the case of an Order of Sources method, the user will always be prompted interactively for the value of the slot to which the method is attached.

Result

When the `NoInherit` operator is used in a condition on the left-hand side of a method, the result produced by the operator is always `TRUE`.

Related Topics

Objects	Methods
Classes	Order of Sources Method
Properties	If Change Method
Actions	Inheritance

NotMember Operator

Definition

The *NotMember operator* is used in the conditions of a rule or method to test whether an object is absent from a given class or list.

Operands

The `NotMember` operator takes two operands:

- The first operand is either a single object or a list of objects specified by a pattern.

- The second operand is either a class or a list of objects specified by a pattern.

The second operand is commonly a list of objects satisfying some qualification or relation, as determined by a prior condition within the same rule or method.

Result

The result produced by the `NotMember` operator is `TRUE` if the first operand is not a member of the class or list designated by the second operand, `FALSE` if it is. If the first operand is a pattern, the condition tests whether at least one of the objects in the corresponding list (for an existential pattern) or all of them (for a universal pattern) are excluded from the second class or list. The contents of the first list are then reduced to the difference of the two (the set of all members of the first that do not belong to the second).

Examples

The following are examples of conditions using the `NotMember` operator:

```
NotMember the_stock Common_Stock
NotMember the_stock <Portfolio>
NotMember <Portfolio> <Common_Stock>
```

Related Topics

Rules	Objects
Methods	Patterns
Conditions	Member Operator

NOW Function

Definition

The *NOW function* is used in expressions to find the current date and time. The expression can appear on the left-hand side or right-hand side of rules and methods.

Syntax

The function consists of the word `NOW` followed by an empty pair of parentheses:

```
NOW( )
```

Argument

The function takes no arguments.

Result

The function returns a date result equal to the current calendar date and clock time at the time of call.

Example

The following is an example using the `NOW()` operator:

```
NOW() = Jul 17 1990 15:22:24
```


Related Topics

Expressions	DAY Function
DATE Function	HOUR Function
TIME Function	MINUTE Function
Patterns	SECOND Function
Interpretations	WEEKDAY Function
YEAR Function	YEARDAY Function
MONTH Function	

Objects

Definition

An *object* is the fundamental representation unit in the Rules Element which can have associated with it one elementary data value or a list value expressed as a string (multi-value).

Structure

Every object has a name, which must comply with the Rules Element's standard rules for a well-formed identifier. The object's information content consists of its *properties* and its *components*. The object may be defined to belong to one or more *classes*, which determine the names and types of its properties.

Properties

The property is always a simple data value belonging to one of the six elementary data types (integer, floating point, boolean, string, date, or time), and is identified by name. Its current value is denoted by appending the name of the property to the name of the object, separated by a period (.). This construction is known as a slot:

```
object_name.property_name
```

In addition, one elementary data value or a list value expressed as a string (multi-value) may be associated directly with the object itself. This value is assigned to a special property named `Value`, which usually need not be named explicitly when referring to the value. For example, the name `object_name` by itself, without any qualifying property, is equivalent to the expression `object_name.Value` when used in places where a slot is expected. If the object is specified in an `@V()` interpretation or in the case where the property name is ambiguous, you will need to use the full construction `object_name.Value`.

Components

Unlike a property, a component (also called a subobject) is in turn a full-fledged object with properties and components of its own. Components need not be (and in general aren't) of the same class as the parent object to which they belong.

Methods

A method is by definition triggered through a message sent directly to the object to which the method is attached. In the case where the system tries to bind a message with a method but the object has no method attached, the system will try to use downward inheritance to obtain one. In a situation where the object belongs to multiple classes, each with its own method defined, then an `InhMethod` operator can be used to resolve the conflict by explicitly naming the parent class.

Creation

Objects can be created by several means:

- Explicitly, via the `New` or `Copy` command in the Object editor.
- Implicitly, by using a previously undefined object name in a condition or action of a rule or method, or as a component of another object.
- Dynamically, by executing the `Retrieve` operator to bring in database information in the course of evaluating a rule or method.
- Dynamically, by executing the `CreateObject` operator in the course of evaluating a rule or method. It also has an equivalent Rules Element application programming interface routine (`NXP_CreateObject`) and Rules Element Execute Library routine (`CreateObjects`).

Objects created dynamically are called dynamic objects. Such objects are temporary, existing only for the duration of the session in which they are created.

Deletion

Objects can be destroyed in either of two ways, depending on how they were originally created:

- Objects created interactively by the application developer, either explicitly or implicitly, are destroyed with the `Delete` command in the Object editor.
- Dynamic objects can be deleted by executing the `DeleteObject` operator in the course of evaluating a rule or method.

Dynamic objects are destroyed automatically by the `Quit` or `Restart Session` command ending the session in which they are created.

Related Topics

Classes	<code>CreateObject</code> Operator
Properties	<code>DeleteObject</code> Operator
Identifiers	Value Property
Data Types	Patterns
Rules	Methods
Dynamic Objects	Slots

Refer to the Intelligent Rules Element Database Integration Guide for information about creating objects using database retrieve operations.

Order of Sources Method

Definition

The *Order of Sources* is an optional system method that can be attached to a slot (property associated with a class or object), defining the procedure for determining the slot's value when needed in the course of evaluating a rule or method. If no Order of Sources method exists, the inference engine uses the system default procedure instead, except in the case of private slots, whose Order of Sources method must be explicitly triggered.

Structure

The method consists most importantly of a sequential list of actions, similar to those on the right-hand side of a rule. If desired, the Order of Sources method can be structured exactly like a rule including a list of conditions on the left-hand side and two separate consequent lists of actions on the right-hand side. The conditions list is optional. Like all methods, the Order of Sources method has no hypothesis component.

Creation

The Order of Sources system method is specified via the Method editor. Creation begins by selecting the `Method` field and displaying the local popup menu for the edit line. Choose the `Select Method` option to view the selection dialog. Select the option `*OrderOfSources` from the list (the asterisk in front of the name distinguishes it from user-defined methods). Or you can also type the name "OrderOfSources" (one word) in the edit line for the `Method` field. The structure to which the method is attached is specified in the `Attach To` field. The structure you specify can be a slot, a class, or an object.

Deleting a user-defined Order of Sources method, causes the system to use the default behavior described under "Default" below.

Invocation

In the case of a public slot with an Order of Sources attached, the inference engine automatically triggers the method when the value of a slot is needed and is set to `UNKNOWN`. Optionally the method can be explicitly triggered by a `SendMessage` operator during the course of evaluating a rule or other method. This allows the application developer to trigger initialization instead of the inference engine. In the case of a class or object with an Order of Sources attached, the `SendMessage` operator must be used in order to trigger the method, but it will no longer be dependent on the Order of Sources strategy (and will actually be treated as a user-defined method by the inference engine).

If the Order of Sources is triggered automatically, and depending on the current strategy, the system executes each action in sequential order until the value of the slot is found and then stops. In the case of an Order of Sources that is triggered explicitly by a `SendMessage` operator, the system will first determine whether the value of slot has already been determined. If the slot value needs to be determined, and depending on the current strategy, the system executes the Order of Sources actions list in sequential order until the value of the slot is found and then stops. The actions execution behavior can be altered for both types of Order of Sources

(triggered automatically or by a `SendMessage` operator) by setting the global or local Order of Sources strategy to `ON/CONTINUE`. If the Order of Sources is triggered automatically, however, arguments that might have been passed by the `SendMessage` operator are ignored.

In the case of a private slot with an Order of Sources method attached, the system is unable to trigger the method automatically. The application developer is required to use the `SendMessage` operator to explicitly trigger the Order of Sources method of a private slot. The `SendMessage` operator must appear in a method and cannot be used in a rule condition or action.

If no explicit Order of Sources method is specified at the level of the slot, a substitute method will be sought by downward inheritance from an including class, superclass, or parent object as directed by the inheritance strategy currently in effect. See the “Inheritance” section for details.

Inheritance

Order of Sources methods can only be inherited downward (from a class to its instances or subclasses, or from an object to its components), never upward. The search through the parent tree hierarchy is directed by the global inheritance strategy and can be class or object-first and depth or breadth-first. Any explicit Order of Sources method defined at the level of the slot overrides this inheritance behavior; to reincorporate the inheritance behavior as part of such a method, include an explicit call to the `InhMethod` operator as described in the “Default” section below. To prevent the method from being inherited, change the `Public` option to `Private` in the Method editor.

When an inheritance conflict exists between two parent objects or classes at the same level, the application developer can use the `InhMethod` operator to override the default inheritance behavior by specifying the parent object to begin the search. When the inheritance conflict occurs between two slots at the same level, the application developer can also set the inheritance priority of the slots to resolve the conflict. If neither approach is used, by default the system chooses the method attached to the parent whose name appears first in alphabetic order.

Default

If no explicit Order of Sources is specified, the value of an unknown slot is determined by the following sequence of steps:

1. An applicable Order of Sources method is sought by downward inheritance from an including class or parent object. If such an inherited method is found, it is used in place of this default method. (Note that methods can only be inherited downward, never upward.)
2. If the desired value is a boolean and appears as the hypothesis of one or more inference rules, the value is sought by backward chaining to those rules.
3. The needed value itself is sought by downward inheritance from an including class or parent object.
4. The value is sought by upward inheritance from a component object.
5. The user is prompted for the value interactively.

Unless the Order of Sources strategy setting is ON/CONTINUE, this process terminates as soon as any step yields a value for the desired property; any remaining steps are skipped.

Any explicit Order of Sources defined for a slot overrides the default method described above. To reincorporate the default behavior as part of such a method, include the equivalent sequence of operators explicitly within the method:

```
InhMethod
Backward
InhValueDown
InhValueUp
AskQuestion
```

To disable downward inheritability of a particular method, select the Private option in the Method editor for the method definition.

Operators

The following operators can occur in an Order of Sources method defined for a slot:

Assign	Strategy
SendMessage	UnloadKB
CreateObject	RunTimeValue*
DeleteObject	InhValueDown*
Retrieve†	InhValueUp*
Write†	InhMethod
Reset	NoInherit
Show†	Backward*†
Execute	AskQuestion*†
LoadKB	Interrupt

Operators marked by an asterisk (*) may be used to obtain a value, with the exception that AskQuestion, Backward, RunTimeValue, InhValueUp and InhValueDown are available only on the right-hand side of the Order of Sources method.

Operators marked by a cross (†) may not be used in the case of a private slot whose value is being sought by the Order of Sources method attached to the slot.

Strategy

Execution of Order of Sources system methods by the inference engine is normally enabled by default, but can be modified if necessary by changing the global inference strategy:

- Interactively through the Strategy Monitor window (from the Expert menu), by turning off the Order of Sources option.
- Dynamically during the course of inference processing itself, via the Strategy operator in a condition or action of a rule or method, using the @SOURCESON=OFF setting.
- In addition to ON and OFF, a third option ON/CONTINUE forces the system to execute every action in the actions list, even after the value of the slot is found. Unless this option is selected, the system will stop executing the Order of Sources actions once the value is found.

Note: The `SendMessage` operator can be used to explicitly trigger an Order of Sources method. The method triggered by the `SendMessage` operator is not affected by any of the strategy settings and will actually be treated as a user-defined method by the inference engine.

During the inferencing process the system first uses the `Strategy` operator setting to determine the current strategy, however, it is possible to invoke the Strategy Monitor window Order of Sources setting from the `Strategy` operator. This option is provided by the `CURRENT` setting in the `Strategy` operator argument dialog box.

Related Topics

Objects	Inheritance
Classes	Inheritance Strategy
Properties	Inference
Actions	Inference Strategy
Rules	Backward Chaining
Slots	Strategy Operator
Methods	InhMethod Operator
If Change Method	SendMessage Operator

Also see the sections on individual operators by name, as listed above.

Patterns

Definition

A pattern is used in the conditions or actions of a rule or method to refer collectively to all existing instances of a class (including those of subclasses) or all components (subobjects) of an object.

Syntax

A pattern consists of the name of a class or object enclosed between angle brackets (`< . . . >`) or curly braces (`{ . . . }`), optionally qualified by a dot (`.`) and a property name. The brackets or braces may be doubled (`<< . . . >>`), tripled (`<<< . . . >>>`), etc., provided that they are evenly balanced on left and right.

The class name may appear between vertical bars inside the brackets or braces (`<| . . . |>`) to distinguish it from an object name.

Interpretations of the class or object name are valid within patterns. The string that appears in the interpretation is the name of a slot that resolves to a class or object name to which the pattern applies.

Scope

The scope of a pattern is limited to the conditions and actions of the single rule or method in which it appears. Occurrences of the same pattern in other rules or methods are separate and unrelated to the one in question. Objects with private slots are not included in the list resulting from a pattern matching statement. When establishing a pattern the system considers only public slots and ignores any objects whose private slots belong to the same class.

Meaning

A pattern represents a list which is defined at runtime and contains an indefinite number of objects. Any condition or action in which the pattern appears is understood to apply separately to each object in the list. For example, the action

```
Assign      <Rect>.length * <Rect>.width  <Rect>.area
```

independently sets the public slot values for each object that belongs to class `Rect` and has the property `area` equal to `length` times `width`.

Note: You cannot do tests on a pattern without specifying a property or using the property `Value`. Objects whose properties comprise a private slot (specified as a `Meta-Slot` attribute of the slot) are not included in the list of objects generated by the pattern.

Initially, the list consists of all existing instances of the specified class or all components (subobjects) of the specified object. Each time the pattern appears in a condition, the list is reduced to only that subset of its previous contents that satisfy the given condition. Later occurrences of the pattern within the same rule refer only to this reduced list of objects, and may in turn reduce its contents still further. To begin a new list based on the same class or object, use a different number of brackets or braces: for example, the patterns

```
<Rect>
```

and

```
<<Rect>>
```

refer to two independent lists of objects belonging to class `Rect`. Action side lists generated by patterns cannot be reduced further because no tests are performed on the list.

The angle brackets `< . . . >` form an existential pattern, meaning “There exists an object in the list such that . . .” Any condition including such a pattern is `TRUE` if there is at least one instance of the given class or component of the given object that satisfies the condition. For example, the condition

```
<      <Item>.quantity * <Item>.cost      10000
```

is `TRUE` if there is at least one instance of class `Item` for which the product of the properties `quantity` and `cost` is less than 10000.

The curly braces `{ . . . }` form a universal pattern, whose meaning is “For all objects in the list, . . .” In this case, the condition is `TRUE` only if it is satisfied by every instance or component in the list. For example, the condition

```
<      {Item}.quantity * {Item}.cost      10000
```

is `TRUE` if `quantity` times `cost` is less than 10000 for every object belonging to class `Item`. Universal patterns can be used in either condition or action lists, but unlike the existential pattern they cannot generate reduced lists.

Because the action side of a rule or method cannot perform tests, only the universal pattern is meaningful. If angle brackets (existential pattern) are used on the actions side of a rule or method, they will be read by the system

as curly braces (universal pattern) and the list will contain all of the objects of the parent on which the pattern is done.

Evaluation

The system completes the evaluation of the entire pattern before it produces a consequence effect on the rule or method condition. This means each public slot specified by a pattern is evaluated before returning the value of the condition. In the case of existential patterns, the evaluation continues even after the system finds one slot that satisfies the condition. In the case of universal patterns, the evaluation stops after one slot fails to satisfy the condition. However, you can assign inference priorities to individual object slots and force the evaluation order of the object slots in the pattern. If no priorities are specified, the default is to process the object slots in alphabetic order.

If the pattern is performed on a set of objects whose properties comprise only private slots, the pattern is not evaluated and the condition is automatically set to `FALSE`. The occurrence of a private slot in the class specified by a pattern will send a message to the Rules Element Transcript.

Implicit Definition

If the name appearing between brackets or braces is not yet known to the system, it will be defined implicitly as a result of its use in a pattern. By default, the name is assumed to refer to a single object rather than a class; to define a new class implicitly, enclose the name between vertical bars inside the brackets or braces. For example, if the name `Item` is not yet defined, the pattern

```
<Item>
```

will prompt you as to whether you want to create an object or class named `Item`, while

```
<|Item|>
```

will create a new class by that name.

Examples

The following are examples of **valid** patterns:

<code><Switch></code>	Existential pattern.
<code>< Switch ></code>	Existential pattern with explicit class name.
<code>{Switch}</code>	Universal pattern.
<code>{\SwitchClassName\}.on</code>	Universal pattern with interpretation to get class name.
<code><Switch>.on</code>	Existential pattern on class members with prop "on".
<code><<Switch>>.on</code>	Produces new existential pattern list for class members with prop "on".

The following are not legal:

<code><Switch</code>	Unbalanced brackets.
<code><<Switch></code>	Unbalanced brackets.
<code><Switch}</code>	Mismatched brackets.
<code>< Switch</code>	Mismatched brackets.

Some additional illegal comparisons using patterns are:

=	<Switch>.on <RefSwitch>.off	Comparison on different classes.
=	<Switch>.on {Switch}.status	Comparison on different pattern types.
=	{Switch}.on <Switch>.status	Comparison on different pattern types.

Related Topics

Objects

Classes

Properties

Rules

Slots

Methods

Conditions

Actions

Data Validation Attribute

POW Function

Definition

The *POW function* is used in expressions to raise a floating point number to any required power. The expression can appear on the left-hand side or right-hand side of rules and methods.

Syntax

The function consists of the word `POW` followed by two arguments in parentheses:

```
POW(x, y)
```

Arguments

Each argument may be any expression yielding a numerical result. The expressions may include patterns or interpretations. If the value of the second argument is not a whole number, the first argument must be greater than or equal to 0.0.

If the value of either argument expression is an integer, it will be converted to an equivalent floating point value.

Result

The function returns a floating point result equal to the first argument raised to the power specified by the second (x^y). The function is equivalent to the expression

```
EXP(y * LN(x))
```

If either argument expression does not produce a numerical value, an error message is posted and the function result is `NOTKNOWN`.

Examples

The following examples illustrate the results of the `POW` function:

```
POW( 3, 5) = 243.0
POW(-3, 5) = -243.0
POW( 3.1, 5.4) = 450.14
POW( 3, -2) = 0.11
POW( 3, 0.5) = 1.73
```

```
POW( 3, 0) = 1.00
POW( 0, 3) = 0.00
```

Related Topics

Expressions	Interpretations
Floating Point Constants	EXP Function
Integer Constants	LN Function
Patterns	

PROD Function

Definition

The *PROD function* is used in expressions to find the product of a set of numerical values. The expression can appear on the left-hand side or right-hand side of rules and methods.

Syntax

The function consists of the word `PROD` followed by any number of arguments in parentheses:

```
PROD(x1, x2, . . . , xn)
```

Arguments

Each argument may be any expression yielding a numerical or time-valued result. There may be either a list of arguments or a pattern matching list.

If some of the argument values are integers and some floating point, the integers will be converted to equivalent floating point values before computation.

Result

The function multiplies together all the argument values and returns their product. For arguments that include patterns, it multiplies all values in the corresponding list.

If any argument is of a non-numeric type, an error message is posted and the function result is `NOTKNOWN`.

Examples

The following examples illustrate the results of the `PROD` function:

```
PROD(365, 240, 577) = 50545200
PROD(98.6, 37.0, -273.18) = -996615.27
PROD(12, 11.7) = 140.4
PROD(TIME(8, 4, 23), TIME(3, 6, 11)) = NOTKNOWN
PROD(123, "456") = NOTKNOWN
```

If class `Tank` has four instances with `capacity` values of 6.3, 14.5, 12.9, and 9.0, then

```
PROD(<Tank>.capacity) = 10605.73
```

Related Topics

Expressions
Data Types
Patterns

Interpretations
SUM Function

Prompt Line Attribute

Definition

The *prompt line attribute* associated with a public slot specifies the text (up to 2,048 characters, extendable via `NXP_BUFSIZE`) to be displayed on the screen when requesting the slot's value interactively from the end user.

Usage

The text of the prompt line is displayed either in the Rules Element main window or a custom window that you provide, whenever the value of the given public slot must be requested from the user. There are several ways this can happen:

- The user explicitly volunteers the value with the `Volunteer` or `Suggest/Volunteer...` command.
- An `AskQuestion` operator is executed in an Order of Sources method.
- The value is needed in the course of evaluating a rule's conditions during inference processing.

The prompt line can only be used in the case of a public slot. Private slots cannot be updated directly and must use a method to determine the slot value.

Creation

The prompt line attribute is specified or edited by typing into the box labeled `Prompt Line` in the Meta-Slot editor. You can also use the `@V()` and `@SELF` constructions in the Prompt Line.

Default

If no prompt line is explicitly specified, one of the following messages will be used by default, depending on the situation:

```
What is the capacity of tank_3?
Volunteer the capacity of tank_3
Modify the capacity of tank_3
```

(where, in this case, the data item being requested is named `tank_3.capacity`).

Inheritance

Inheritability of the prompt line attribute is controlled by the inference engine. If no prompt line has been specified for the slot, the system will try to inherit the prompt line attribute of the slot's parent class or object.

Related Topics

Objects	Order of Sources Method
Classes	Inference
Properties	Meta-Slots
Rules	AskQuestion Operator
Conditions	Question Window Attribute
Methods	SELF
Slots	

Properties

Definition

A *property* is an attribute which can be associated with an object or class.

Form

Every property has a name, which must comply with the Rules Element's standard rules for a well-formed identifier. Its value is always a simple data value belonging to one of the six elementary data types (integer, floating point, boolean, string, date, or time).

Scope

The definition of a given property is not local to a particular object or class, but global throughout the entire system. This means that two objects may not have properties with the same name but different types: a given property name always designates a value of the same type, wherever it may occur. (The specific value of the property may, of course, vary from one object to another.) The one exception to this rule is the special, predefined property named `Value`; see "Value Property" for more information.

Creation

Properties can be created in either of two ways:

- Implicitly, by using a previously undefined property name in a condition or action of a rule or method in the Rule, Object, Class or Method editor.
- Interactively, via the `New` or `Copy` command in the Property editor.

Deletion

Properties are destroyed with the `Delete` command in the Property editor.

Access

The current value of a property when associated with a given object or class is denoted by appending the name of the property to the name of the object, separated by a period (`.`). This construction is known as a slot:

```
object_name.property_name
```

Slots can be defined by a meta-slot attribute to be either public or private. A public slot's current value can be changed in either of two ways:

- Explicitly, by executing the `Assign` operator in a condition or action of a rule or method.
- Interactively, via the `Volunteer` command.

A private slot's current value can be changed only by triggering a method attached to the slot. Private slots let you use object-encapsulation and therefore are accessible only by methods.

Related Topics

Objects

Classes

Slots

Data Types

ConditionsRules

Value PropertyData ValidationAttribute

Assign OperatorMethods

Question Window Attribute

Definition

The *question window attribute* associated with a public slot specifies the window to be displayed on the screen when requesting the slot's value interactively from the end user. The window is a custom resource created using the GUI builder, provided with the Open Interface Element.

Usage

The window you specify in the question window attribute lets you use a question window of your own design instead of the session control panel of the Rules Element main window. The custom window is opened during application processing by the question handler:

- For a public slot with a window specified in the slot's meta-slot.
- For a public slot with a window specified in the meta-slot of one of the slot's parents.

The question window can only be used in the case of a public slot. Private slots cannot be updated directly and must use a method to determine the slot value.

Creation

The question window attribute is specified by typing the name of the window into the box labeled `Question Win` in the Meta-Slot editor. The name must include the window's full resource name:

`ModuleName.WindowName`

The window resource itself is created through the Resource Browser window as described in the *Open Interface Element User's Guide*.

Default

If you are running your application from the Rules Element development version and no question window is explicitly specified, the system displays the question in the session control panel of the Rules Element main window that uses the meta-slot prompt line attribute or default question to solicit the value of the slot with a list of choices for string slots.

If, however, you want to run your application using the Rules Element standalone and no question window is explicitly specified, the system does not have the option to display the session control panel (since there will be no main window). Consequently, the user prompt will never be displayed and the system automatically assigns the value `NOTKNOWN` to the slot value. Before running a standalone application, assign a simple window to the question window attribute for every slot that you anticipate may become evaluated.

Inheritance

Inheritability of the question window attribute is controlled by the inference engine. If no question window has been specified for the slot, the system will try to inherit the question window attribute of the slot's parent class or object.

Related Topics

Methods

Prompt Line Attribute

Meta-Slots

Slots

For complete details about building graphical user interfaces for your Rules Element application, refer to the *Open Interface Element User's Guide*.

RAND Function

Definition

The *RAND function* is used in expressions to generate a random floating point number. The expression can appear on the left-hand side or right-hand side of rules and methods.

Syntax

The function consists of the word `RAND` followed by an empty pair of parentheses:

```
RAND ( )
```

Arguments

The function takes no arguments.

Result

The function returns a random floating point result generated from a uniform distribution on the range $0 \leq x \leq 32767$. The floating point number will never have a decimal part.

Examples

The following examples illustrate the results of the `RAND` function:

```
RAND() = 17515.0
RAND() = 542.0
RAND() = 26874.0
```

Related Topics

Expressions

Floating Point Constants

RANDOMSEED Function

RANDOM Function

RANDOMMAX Function

RANDOM Function

Definition

The *RANDOM function* is an alternate way to generate a random floating point number in expressions. On some platforms (usually UNIX) it is better than `RAND` due to the specific machine implementation, while on others it is exactly the same as `RAND`. The expression can appear on the left-hand side or right-hand side of rules and methods.

Syntax

The function consists of the word `RANDOM` followed by an empty pair of parentheses:

```
RANDOM( )
```

Arguments

The function takes no arguments.

Result

The function returns a random floating point result which does not include a decimal part. Note that this result is more random than the `RAND` function on many platforms. If an argument is given to `RANDOMSEED`, then the argument is used as the seed for the random number generator.

`RANDOMMAX` is the maximum value over which the uniform distribution is distributed. Note that `RANDOMMAX` is machine dependent which means that the range of the `RANDOM` function is machine dependent and thus applications using it may not behave exactly the same from one hardware platform to the next. However, the ratio `RANDOM() / RANDOMMAX()` provides you with a random generator that is portable across platforms. It returns a floating point value between 0 and 1.

Examples

The following examples illustrate the results of the `RANDOM` function:

```
RANDOM() = 5758.0
RANDOM() = 247512.0
```

Related Topics

Expressions	RAND Function
Floating Point Constants	RANDOMMAX Function
RANDOMSEED Function	

RANDOMMAX Function

Definition

The *RANDOMMAX function* is used to get the upper bound of the `RANDOM` function. The function can appear on the left-hand side or right-hand side of rules and methods.

Syntax

The function consists of the word `RANDOMMAX` without any arguments:

```
RANDOMMAX ( )
```

Arguments

This function takes no arguments.

Result

This function returns the upper bound over which the `RANDOM` function will generate uniform random numbers. The upper bound is machine dependent ($2^{31}-2$ on the Macintosh, $2^{31}-1$ on the UNIX platforms, and $2^{15}-1$ on the DOS machines).

Examples

The following examples illustrate the results of the `RANDOMMAX` function when used on the Macintosh (it returns the value of $2^{31}-2$):

```
RANDOMMAX ( ) = 2147483646
```

Related Topics

Expressions	RAND Function
Floating Point Constants	RANDOM Function
RANDOMSEED Function	

RANDOMSEED Function

Definition

The *RANDOMSEED function* is used to give a specific seed to the `RANDOM` random number generator. On machines where `RANDOM` and `RAND` are identical (typically non-UNIX), `RANDOMSEED` will also seed the `RAND` function. The function can appear on the left-hand side or right-hand side of rules and methods.

Syntax

The function consists of the word `RANDOMSEED` followed by the seed within a pair of parentheses:

```
RANDOMSEED(x)
```

Arguments

The function takes an integral argument. The argument can be any slot or interpreted value which evaluates to an integer.

Result

Giving the `RANDOMSEED` function a particular value within an application is useful for generating the same sequence of random numbers for each run of the application. The function returns the integer argument.

Examples

The following example illustrates the results of the `RANDOMSEED` function:

```
RANDOMSEED(12345) = 12345
```

Related Topics

Expressions	RAND Function
Floating Point Constants	RANDOM Function
RANDOMMAX Function	

Reserved Words

Definition

A reserved word is a word that is used by the Rules Element for a special purpose (such as the name of a type, operator, or special value) and is not available for use as an ordinary identifier. Some reserved words are case sensitive, others are case insensitive.

The following words are reserved:

AND*	InhMethod	Retrieve
AskQuestion	InhValueDown	RunTimeValue
Assign	InhValueUp	SELF*
Backward	INTEGER*	SendMessage
BOOLEAN	Interrupt	Show
CreateObject	KNOWN*	Strategy
DATE	LoadKB	STRING
DEFAULT*	Member	Time
DeleteObject	No	TRUE*
Execute	NoInherit	UNKNOWN*
FALSE*	NOTKNOWN*	UnloadKB
FLOAT	NotMember	Value*
IfChange*	Null*	Write
	OR	Yes

* denotes case insensitive reserved word.

Examples

The following examples show the difference between case sensitive and case insensitive reserved words:

UNKNOWN	reserved (case insensitive)
uNknoWn	reserved (case insensitive)
Yes	reserved (case sensitive)
YES	not reserved (case sensitive)

Related Topics

Identifiers

Expressions

Data Types

Also see the sections on individual operators and functions by name, as listed above.

Reset Operator

Definition

The *Reset operator* is used in rules and methods to reset a variable to UNKNOWN.

Operands

The *Reset operator* takes one operand, which may be either a slot or a list of slots specified by a pattern.

Effects

The designated slot is set to the special value UNKNOWN, denoting a value that has not yet been determined. If the operand includes a pattern, all slots in the corresponding list are set to UNKNOWN.

If the slot to be reset is a hypothesis, all rules and the rules left-hand side conditions pointing to it are reset to the UNKNOWN state as well. The *Reset operator* is then applied in turn to any hypotheses occurring in the conditions of these rules, propagating backward recursively to unlimited depth. Only hypotheses are affected, however; no other data occurring in the conditions of any rule are reset.

If the designated slot is of any type other than boolean, or does not occur as the hypothesis of any rule, then only that one slot is reset to UNKNOWN.

The effects of the *Reset operator* are never propagated forward to other rules and have no effect on the state of the agenda. If there are any *If Change* actions, they will not be fired.

Result

When used in a condition on the left-hand side of a rule, the *Reset operator* always produces a TRUE result unless the operand includes a pattern with no matching values, in which case the result is NOTKNOWN.

Examples

The following are examples of conditions or actions using the `Reset` operator:

```
Reset      total
Reset      customer.name
Reset      all_tanks_full
Reset      tank_9.full
Reset      <Tank>.full
```

Related Topics

Rules	Data Types	Agenda
Methods	Hypotheses	Objects
Conditions	Patterns	Properties
Actions	Forward Chaining	

Retrieve Operator

Definition

The *Retrieve operator* is used in the conditions or actions of rules and methods to read information from a database or spreadsheet.

Operands

The `Retrieve` operator takes two operands:

- The first operand is either a string constant or an interpretation to a string constant specifying the name of the file containing the database to be queried or the login name/password for a DBMS.
- The second operand consists of a series of parameters defining the specific retrieval operation to be performed.

Parameters

The second operand may include the following parameters:

@TYPE	Type of database (creator software and file format)
@BEGIN	Command string for opening transaction
@END	Command string for closing transaction
@QUERY	Command string for querying database
@ARGS	Argument list for query command
@ATOMS	List of objects or properties affected
@NAME	Correspondence between records and objects
@FIELDS	List of field names to retrieve from
@PROPS	List of properties to retrieve to
@SLOTS	List of slots to retrieve to
@FILL	Create new objects

@CREATE	Classes or parents to link new objects to
@UNKNOWN	Retrieve UNKNOWN values
@FWRD	Forward retrieved values
@CURSOR	Current position for sequential retrieval

See the Database Integration Guide for further details on the meaning and use of these parameters.

When entering a `Retrieve` action in the Rule editor or Method editor, clicking in the space for the second operand displays the Database editor dialog box for specifying the retrieval parameters interactively, rather than by explicitly typing them in as listed above.

Note that data retrieved for a private slot named in `@SLOTS` is ignored unless the `Retrieve` operator appears in a method specifically triggered for the slot. See the description of Slots for more information about using private slots.

Effect

The requested information is retrieved from the specified database to the Rules Element knowledge base for further processing.

Result

When used in a condition on the left-hand side of a rule, the `Retrieve` operator always produces a `TRUE` result, even if no records are retrieved satisfying the given query. The only exception is if an error occurs while attempting to open the database or transmit the query, in which case the result is `FALSE`.

Forward Chaining

Actions and conditions in rules and methods involving the `Retrieve` operator can forward chain the new value of the slot to other rules in which the slot appears in a condition (causing the hypotheses of those rules to be placed on the agenda for consideration). This form of forward chaining, known as Forward Action-Effects, is controlled first by a strategy setting in the Database editor. If the `Current` option is checked, the system uses the local strategy currently in effect (determined by the `Strategy` operator), unless the `Retrieve` operator appears in a left-hand side condition, in which case the Rule Global strategy setting in the Strategy Monitor window is used.

Data that belongs to a private slot cannot trigger forward chaining since private slot data cannot appear in the conditions or actions of rules. Only data that belongs to public slots can trigger forward chaining.

Examples

See the Database Integration Guide for examples of the use of the `Retrieve` operator.

Related Topics

Rules	Properties
Methods	Slots
Actions	String Constants
Conditions	Write Operator
Objects	Inference Strategy
Classes	Forward Chaining

Also see the Database Integration Guide for more information on database operations.

ROUND Function

Definition

The *ROUND function* is used in expressions to find the nearest whole number to a given floating point number. The expression can appear on the left-hand side or right-hand side of rules and methods.

Syntax

The function consists of the word `ROUND` followed by a single argument in parentheses:

```
ROUND(x)
```

Argument

The argument may be any expression yielding a floating point result. The expression may include patterns or interpretations.

Result

The function returns a floating point result equal to the nearest whole number to the argument. Notice that although the result is always a whole number, it is of type `FLOAT` rather than `INTEGER`.

If the argument expression does not produce a numerical value, an error message is posted and the function result is `NOTKNOWN`.

Examples

The following examples illustrate the results of the `ROUND` function:

```
ROUND(3.1416) = 3.0
ROUND(98.6)   = 99.0
ROUND(-273.18) = -273.0
ROUND(-9.9)   = -10.0
```

Related Topics

Expressions	Patterns
Floating Point Constants	Interpretations
Integer Constants	<code>CEIL</code> Function
<code>FLOOR</code> Function	

Rules

Definition

A *rule* is the Rules Element's basic unit of inference and reasoning.

Structure

Every rule consists of the following parts:

- One or more conditions under which the rule is to be invoked, or fired.
- Exactly one hypothesis, which is inferred to be true if all of the conditions are satisfied.
- Zero or more Then actions to be taken when the conditions are satisfied.
- Zero or more Else actions to be taken when any condition is not satisfied.

Collectively, the conditions constitute the left-hand side of the rule and the hypothesis and actions together constitute the right-hand side.

In addition, a rule may optionally have an inference priority or inference slot to control its order of evaluation relative to other rules leading to the same hypothesis, and a comment attribute and why attribute to help document its meaning or purpose for the benefit of the application developer.

Evaluation

The evaluation of a rule may be triggered in either of two ways:

- By *backward chaining*, when its hypothesis is suggested as a goal to be investigated.
- By *forward chaining*, when a data value named in one of its conditions is volunteered.

Evaluation proceeds by evaluating each of the conditions on the rule's left-hand side:

- If all the conditions are `TRUE`, the rule's hypothesis is set to `TRUE` and all actions specified on its right-hand side are executed.
- If any condition is `NOTKNOWN`, the hypothesis is set to `NOTKNOWN`.
- Otherwise, if any condition is `FALSE`, the hypothesis is set to `FALSE`.

Conditions and actions are normally executed sequentially, in the order they appear in the rule definition, but this order may be altered by the inference priorities or inference slots of the data involved. Rule evaluation stops as soon as one condition is evaluated as `FALSE`. Depending on the strategy options currently in effect, the inferred value of the hypothesis and the results of any actions taken may be forward-chained, resulting in other hypotheses being placed on the agenda for consideration. Actions may be executed whether or not the rule's conditions are satisfied by specifying separate lists of actions using the Then and Else lists. If all the conditions are met, the system executes the Then actions list; otherwise, the system executes the Else actions list.

Creation

Rules are created interactively via the `New` and `Copy` commands in the Rule editor (you can also create rules by editing the text knowledge base directly). Rules cannot include tests on private data. Only public slot values may be tested in rule conditions. Private slots are accessible by methods only.

Deletion

Rules are always deleted interactively, via the `Delete` command in the Rule editor.

Related Topics

Hypotheses

Conditions

Actions

Slots

Inference Priority Attribute

Inference Slot Attribute

Why Attribute

Agenda

Strategy

Forward Chaining

Backward Chaining

Semantic Gates

Comment Attribute

RunTimeValue Operator

Definition

The *RunTimeValue operator* is used in `Order of Sources` methods to define a default value for a property.

Operand

The `RunTimeValue` operator takes one operand, which must be one of the following:

- A constant of the proper type for the property being initialized.
- The special value `NOTKNOWN`.

Effect

The value of the operand is assigned as the value of the property to which this `Order of Sources` method belongs. This operator is typically used as the last line of the method, to specify a default value for the property in case all preceding actions fail to yield a usable value.

Notice the difference between `RunTimeValue` and the related `Initial Value` attribute from the `Meta-Slot` editor. `RunTimeValue` specifies a default value to be set dynamically during inference processing; the `Init Value` attribute specifies an initial value to be set at system initialization time. In the case of multiple KBs, always use `RunTimeValue` instead of the `Init Value` attribute because the `Init Value` attribute won't be used when a knowledge base is dynamically loaded.

Examples

The following are examples of actions using the `RunTimeValue` operator:

```
RunTimeValue      28
RunTimeValue      -273.18
RunTimeValue      "SHAZAM!"
RunTimeValue      TRUE
RunTimeValue      DATE(1981,6,8,21,8,46)
RunTimeValue      TIME(8,4,23)
RunTimeValue      NOTKNOWN
```

Related Topics

Objects	Methods
Properties	Actions
Data Types	Init Attribute
Value	Order of Sources Method

SECOND Function

Definition

The *SECOND function* is used in expressions to extract the seconds field of a date or time. The expression can appear on the left-hand side or right-hand side of rules and methods.

Syntax

The function consists of the word `SECOND` followed by a single argument in parentheses:

```
SECOND(d)
```

Argument

The argument may be any expression yielding a date or time result. The expression may include patterns or interpretations.

Result

The function returns an integer result equal to the seconds field of the argument. For date arguments, the result ranges from 0 to 59.

If the argument expression does not produce a date or time value, an error message is posted and the function result is `NOTKNOWN`.

Examples

The following examples illustrate the results of the `SECOND` function:

```
SECOND( DATE(1492,10,12) )      = 0
SECOND( DATE(1981,6,8,21,8,46) ) = 46
SECOND( TIME(8,4,23) )         = 23
SECOND( TIME(3,6,11,22,34,17) ) = 17
SECOND( "October 12, 1492" )   = NOTKNOWN
```


Related Topics

Expressions	DAY Function
DATE Function	HOUR Function
TIME Function	MINUTE Function
Patterns	WEEKDAY Function
Interpretations	YEARDAY Function
YEAR Function	NOW Function
MONTH Function	

SELF

Definition

The special name *SELF* is used to refer to the current class or object executing a method, data validation function, or prompt line attribute.

Syntax

The name *SELF* is case-insensitive. Typically, it is qualified by a dot (.) and a property name to refer to a specific slot. All of the following forms are equivalent:

```
SELF.property_name
Self.property_name
self.property_name
sELF.property_name
```

It is important to realize that *SELF* is not usually used to designate an object slot without a corresponding property name. The only exception is in the case of a hypothesis slot since the property name *VALUE* need not be explicitly stated for hypotheses.

Usage

The name *SELF* may be used in a Data Validation function, any action occurring in a method (but not in a rule), or within the *@V* syntax of the Prompt Line attribute. In a method associated with a property of a class, it refers to the particular object (instance) of the class for which the method is being executed.

SELF is instantiated by the value of the current object under evaluation. Since Prompt Lines, Data Validation attributes, and methods can be inherited down, the child object inheriting the item instantiates the *SELF* variable.

If the dynamic quality of the *SELF* is desired in the *@STRING* parameter of an execute routine, the syntax is *@SELF*.

The *SELF* keyword must be used when referring to private slots in a method associated with the slot.

Example

A class named `Rectangle` might include the following action in an *If Change* method associated with property `width`:

```
Assign SELF.width * SELF.height SELF.area
```

If `theBox` is an instance of class `Rectangle` whose `width` property is changed in the course of inference processing, the `If Change` method will be executed with `SELF` referring to object `theBox`. The action shown above will then set `theBox.area` to the product of `theBox.width` and `theBox.height`.

The method action:

```
Assign      10      <self>.prop
```

will set the value of the slot `<self>.prop` of all children of the current object or class with the property `prop` to 10.

Related Topics

Objects	Methods
Classes	Order of Sources Method
Properties	If Change Method
Actions	Prompt Line Attribute
Rules	Data Validation Attribute
Slots	

Semantic Gates

Definition

A *semantic gate* (also called a *strong link*) is a connection between the left-hand side conditions of two inference rules that share the same data.

Creation

Semantic gates are created implicitly by defining rules that share data in the relevant ways; no special action is required to establish them.

Deletion

Just as semantic gates are not explicitly created, they cannot be explicitly destroyed except by deleting the rules involved, or by redefining them so as to remove the relevant data dependencies.

Operation

Each time a data item or pattern is evaluated in the course of inference processing, the Rules Element searches the knowledge base for other rules whose conditions refer to that same data. For each such rule, it evaluates the relevant condition and, if `TRUE`, places the rule's hypothesis on the agenda for later consideration. When this hypothesis comes to the top of the agenda, its value will be sought by backward chaining. Notice that this can trigger the evaluation of all rules leading to the given hypothesis, not only those that refer to the original data item.

Data associated with private slots cannot form semantic gates because private slots cannot appear in rule conditions. Only public slots that appear in rule conditions can form semantic gates.

Precedence

Hypotheses generated as a result of semantic gates have lower precedence (and consequently are placed lower on the agenda) than those generated by backward chaining, but higher than those generated via context (weak) links. When several hypotheses are placed on the agenda via gates, their precedence is determined according to the inference priorities of the rules involved.

Strategy

The use of semantic gates is normally enabled by default, but can be disabled if necessary by changing the global inference strategy. This can be done in either of two ways:

- Interactively through the Strategy Monitor window (from the Expert menu), by turning off the `Forward through Gates` option.
- Dynamically during the course of inference processing itself, via the Strategy operator in the conditions or actions of a rule, using the `@PTGATES=OFF` setting.

During the inferencing process the system first uses the Strategy operator setting to determine the current strategy, however, it is possible to invoke the Strategy Monitor window's `Forward through Gates` setting from the Strategy operator. This option is provided by the `CURRENT` setting in the Strategy operator argument dialog box.

Related Topics

Objects	Inference
Properties	Agenda
Classes	Backward Chaining
Rules	Forward Chaining
Conditions	Inference Priority Attribute
Actions	Inference Slot Attribute
Hypotheses	Inference Strategy
Patterns	Strategy Operator
Slots	Context Links

SendMessage Operator

Definition

The *SendMessage operator* is used in the conditions and actions of rules and methods to explicitly trigger user-defined methods and pass arguments that the method uses in its conditions and actions.

Operands

The `SendMessage` operator takes two operands:

- The first operand is a quoted string specifying the name of the method to be triggered.
- The second operand requires the name of one or more addressees which will receive the message to trigger a method. It can be a class,

object, slot, or property name. As an alternative it can also be a pattern match when a list of addresses belongs to the same class. (The method to trigger need not be attached directly to the target object since methods can be inherited.)

Interpretations cannot be specified for the addressee using either the @V or \obj.prop\ notation.

- Optionally, the second operand can include a series of *message passing parameters* specifying the arguments to pass to the method.

Parameters

The second operand may include the following message parameters:

@TO= ;	Name of addressee(s) to send the message to. Can be a class, object, slot, or property name. List of addressees must be separated by commas or the list can be specified by the desired pattern matching syntax. Or, can be a pattern match on a class (i.e., <Figures>).
@ARG1= ;	Corresponds to the first argument to pass to the addressees, can be a value you supply or a slot name. (Optional)
...	
@ARGx= ;	Corresponds to the last argument to pass to the addressees, can be a value you supply or a slot name. (Optional)

The order of the arguments list determines which variable it corresponds to in the Method's local argument definition template. The first argument corresponds to the first row of the Method editor's Local Arguments component, the second argument corresponds to the second row of the Method editor's Local Arguments component, and so on. See the Message dialog window below for more details about specifying local arguments.

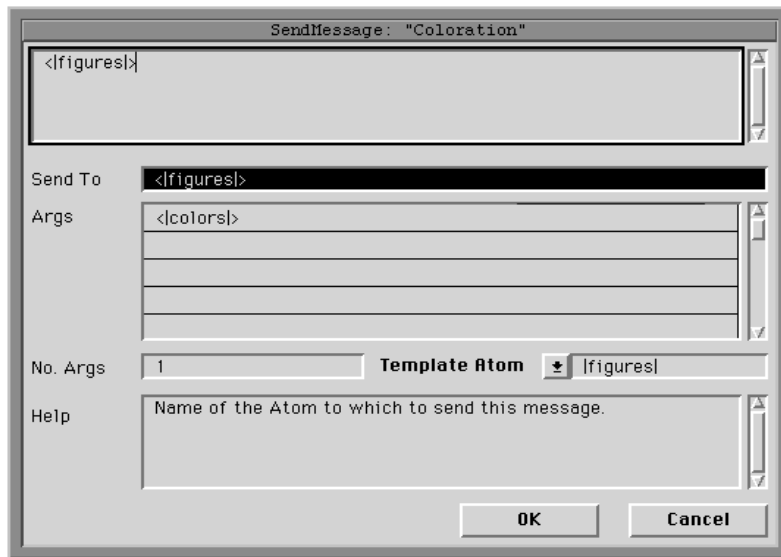
If a slot name is used an argument to pass, it is usually a public slot. Private slots can also be used as arguments but have the particular restrictions that they can appear only in the method attached to the private slot and they can only be passed by value (not by reference). The SELF keyword must be used to refer to the private slot.

Message Dialog

When entering a SendMessage condition or action in the Rule editor or Method editor, clicking in the space for the second operand displays a special dialog box for specifying the addressee(s) and optional message passing parameters interactively, rather than by explicitly typing in the keywords listed above. The SendMessage dialog window has the following fields:

Send To	This field holds the names of one or more addressees. A list can be specified by pattern matching syntax or individual atoms separated by commas. No quotes are needed; the system inserts them automatically.
---------	--

Template Atom	This menu button is used with the Args table (see below). It lets you choose a prototype for your arguments (argument prototypes are defined in the Method editor Local Arguments area). The displayed list of possible prototypes is limited to object structures that have the method named as the first operand attached.
Args	Each row of this table corresponds to a single argument to pass to the method. The Template Atom selection helps you to identify the order that the defining atom expects the arguments. Click in a row and the system displays the argument parameters (Name, Type, and Nature) from the Method editor's argument template in the Help box. .



Send To list must be able to use the same argument prototype (specified by the Template Atom field). Sending messages to addressees with local arguments that are defined differently requires separate SendMessage operators. If arguments that are passed during application processing do not match types, the system writes an error message to the Transcript window and automatically sets a condition with the SendMessage operator to FALSE. If more arguments are passed to the method's local variables than needed, the extra arguments are ignored. If desired, passing arguments to local variables can be avoided by defining an initial value in the Method editor for each local variable used by that method.

Effect

The method named as the first operand is triggered for the list of addressees specified by the second operand. If no method is specified at the level of an addressee, a substitute method of the same name will be sought by downward inheritance from an including class or parent object as directed by the inheritance strategy currently in effect. If the message is sent to a slot, the system can also try to trigger the method attached to the property of the

slot, or to the property of the same name (a property that exists independent of an object or class)

The application developer can resolve inheritance conflicts between two parent slots by assigning inheritance priorities to the slots or through the `InhMethod` operator to explicitly name an inheritance path. If no conflict resolution is specified, the method is chosen by default based on the alphabetic order of the parent names.

The method that is successfully triggered is treated by the system as a “user-defined” method whether it was originally created as a user-defined method or as a system method (Order of Sources or If Change). In the default strategy case, the system executes the list of Then or Else actions, depending on the evaluation status of the method left-hand side, from top to bottom until the value of the slot is found. Changing the local or global Order of Sources strategy to ON/CONTINUE will force the system to execute all the actions in the list even after the value of the slot is found.

The triggered method may receive data to be used as local arguments in its list of conditions and actions. The `SendMessage` operator specifies the data, and the triggered method processes the data according to a template that defines its usage. Data passed to the object receiving the message can be passed by reference or by value as defined in the `Local Arguments` component of the Method editor.

When the data is a slot value “passed by reference,” the method’s actions list can alter the value of the slot that contains the data and may produce forward chaining action effects and trigger any If Change method attached to the slot. Public slots can be passed by reference but private slots cannot (private slots value are accessible only locally by the method). When data is “passed by value,” the value is used by the method locally and has no side-effects on processing. Both private and public slots can be passed by value. If a slot name is specified as an argument but no initial value appears in the Meta-Slot editor, the system will use the default value specified in the Method editor arguments template.

The `Nature` field of the `Local Arguments` component in the Method editor lets you specify how the argument is passed to the named method: select the `SlotRef` popup menu option when you want the slot value to be passed to the method by reference (thus allowing the method to modify the named slot), or select the `Slot` popup menu option when you want the slot value to be passed to the method by value (thus preventing named slot from being modified outside of the method). Note: If the argument passed is an object or class name, it is always passed by Reference (never by Value). For more information about the Method editor, refer to the User’s Guide.

Result

When the `SendMessage` operator is used in a condition on the left-hand side of a rule or method, the result produced by the operator is `TRUE` if the message is successfully bound to the method, `FALSE` if the named method does not exist at the level of the addressee or at its parent object level or if arguments passed during application processing do not match the types specified for the method’s local variables.

Examples

The following are examples of actions using the `SendMessage` operator:

```
SendMessage "Init"           @To=<|Figures|>
SendMessage "Rotate"        @To=<|Figures|>, @Arg1=90
SendMessage "ComputeArea"   @To=Circle
SendMessage "ComputeArea"   @To=circle1, @Arg1=circle1.radius
SendMessage "Close"         @To=valve1, valve2, valve3
```

It is not legal to use interpretations in the arguments list of the `SendMessage` operator.

Refer to the User's Guide for information about implementing the method and its local arguments.

Related Topics

Objects

Properties

Conditions

Actions

Rules

Slots

Methods

InhMethod Operator

Inheritance Priority

Inheritance Strategy

Pattern Matching

Show Operator

Definition

The *Show operator* is used in rules and methods to display the contents of an information file on the screen for the benefit of the user. It has the same functionality as the Apropos command from the Rules Element's pop-up menus.

Operands

The Show operator takes one or two operands:

- The first operand is a string constant or an interpretation evaluating to a string constant (using the `@v(object.prop)` syntax) specifying the name of the file containing the information to be displayed. It must be between double quotes.
- The optional second operand consists of a series of display parameters controlling the display of the information.

File formats

The file-name extension indicates the type of information the file contains and the form in which it is encoded. You are not required to specify the extension since the Rules Element recognizes the type by reading the file. The following file formats are recognized:

<code>.nbm</code>	Rules Element bitmap file on Unix or VMS (was <code>.bmap</code>)
<code>.bm</code>	X Windows bitmap file (was <code>.x</code>)
<code>.bmp</code>	PC bitmap file
<code>.gif</code>	Giff format file
<code>.mcp</code>	MacPaint file
<code>.txt</code>	ASCII text file

Additional formats supported on the Macintosh include PICT (drawing, eg. MacDraw) and PICT2.

Of these file types, all but the PC bitmap file (`.bmp`) are portable across platforms. You can use the Rules Element-provided converter utility when porting to another platform, as described in your Installation Guide.

If the file name has no extension, the Rules Element will try all possible extensions. This allows the same knowledge base to run easily on different platforms. To convert to a non-graphic terminal, for example, you can simply replace your graphics files with text files (`.txt`) without modifying the knowledge base itself.

You can specify a list of directory names which will be searched automatically for the designated information files if the full pathname is not given by:

- Using the SearchPaths string resource on the Macintosh
- Setting the ND_DATA environment variable under Unix, VMS, and PCs.

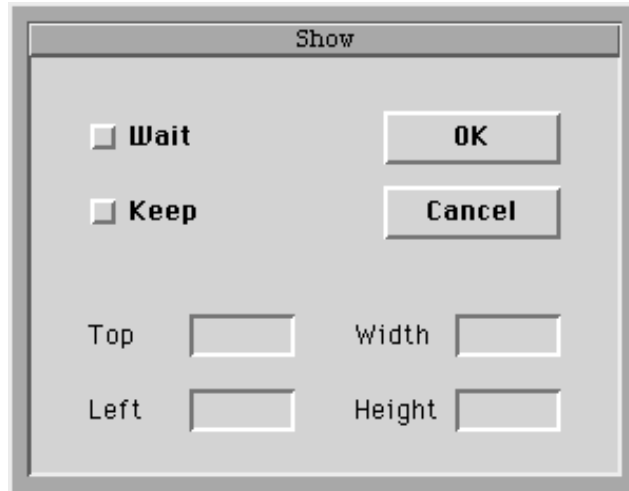
Parameters

The second operand may include the following display parameters:

<code>@KEEP=TRUE ;</code>	Display information in a new window and keep it until the next show or the user explicitly closes it.
<code>@KEEP=FALSE ;</code>	Use same window as previous Show operation
<code>@WAIT=TRUE ;</code>	Display Continue and Close buttons; wait for mouse click before continuing
<code>@WAIT=FALSE ;</code>	No Continue button; just display information and continue processing
<code>@RECT=<i>left , top ;</i></code>	Specify window's location
<code>@RECT=<i>left , top , width , height ;</i></code>	Specify window's location and size

Show Dialog

When entering a Show condition or action in the Rule editor or Method editor, clicking in the space for the second operand displays a special dialog box for specifying the display parameters interactively, rather than by explicitly typing in the keywords listed above:



Effect

The information contained in the file named as the first operand is displayed in a window on the screen, subject to the display options specified by the second operand.

Result

When used in a condition on the left-hand side of a rule or method, the Show operator always produces a TRUE result, even if no information file exists with the specified name.

Note: The Show operator can be customized by installing an APROPOS handler with the Rules Element application programming interface. The Rules Element will use the user-defined function instead of the default behavior described above.

Examples

The following are examples of actions using the Show operator:

```
Show "Diagnostic1"
Show "Diagnostic1.mcp"
Show "Diagnostic1.txt"
Show "Diagnostic1" @KEEP=FALSE;@WAIT=TRUE;
Show "Diagnostic1" @KEEP=TRUE;@WAIT=TRUE;@RECT=100,150;
Show "@v(obj.prop)"
@KEEP=TRUE;@WAIT=FALSE;@RECT=100,150,275,140;
```

Related Topics

Rules

Methods

Conditions

Actions

Apropos handler of API

SIGN Function

Definition

The *SIGN function* is used in expressions to find the sign of a number. The expression can appear on the left-hand side or right-hand side of rules and methods.

Syntax

The function consists of the word `SIGN` followed by a single argument in parentheses:

```
SIGN(x)
```

Argument

The argument may be any expression yielding a numerical result. The expression may include patterns or interpretations.

Result

The function returns an integer result equal to the sign of the argument:

- If the argument is positive, the function result is 1.
- If the argument is zero, the function result is 0.
- If the argument is negative, the function result is -1.

If the argument expression does not produce a numerical value, an error message is posted and the function result is `NOTKNOWN`.

Examples

The following examples illustrate the results of the `SIGN` function:

```
SIGN(28)      = 1
SIGN(-5)     = -1
SIGN(98.6)   = 1
SIGN(-273.18) = -1
SIGN(0)      = 0
```

Related Topics

Expressions

Floating Point Constants

Integer Constants

Patterns

Interpretations

SIN Function

Definition

The *SIN function* is used in expressions to find the sine of a floating point number. The expression can appear on the left-hand side or right-hand side of rules and methods.

Syntax

The function consists of the word `SIN` followed by a single argument in parentheses:

```
SIN(x)
```

Argument

The argument may be any expression yielding a numerical result. The expression may include patterns or interpretations.

If the value of the argument expression is an integer, it will be converted to an equivalent floating point value.

Result

The function returns a floating point result equal to the sine of the argument. The argument is assumed to be expressed in radians.

If the argument expression does not produce a numerical value, an error message is posted and the function result is `NOTKNOWN`.

Examples

The following examples illustrate the results of the `SIN` function:

```
SIN( 0.0)      = 0.0
SIN( 3.14 / 6) = 0.5
SIN( 3.14 / 2) = 1.0
SIN( 3.14)     = 0.0
SIN(-3.14 / 2) = -1.0
```

Related Topics**Expressions**

Floating Point Constants

Integer Constants

Patterns

Interpretations

`COS` Function

`TAN` Function

`ASIN` Function

SINH Function

Definition

The *SINH function* is used in expressions to find the hyperbolic sine of a floating point number. The expression can appear on the left-hand side or right-hand side of rules and methods.

Syntax

The function consists of the word `SINH` followed by a single argument in parentheses:

```
SINH(x)
```

Argument

The argument may be any expression yielding a numerical result. The expression may include patterns or interpretations.

If the value of the argument expression is an integer, it will be converted to an equivalent floating point value.

Result

The function returns a floating point result equal to the hyperbolic sine of the argument.

If the argument expression does not produce a numerical value, an error message is posted and the function result is NOTKNOWN.

Examples

The following examples illustrate the results of the `SINH` function:

```
SINH( 0.0 ) = 0.0
SINH( 0.5 ) = 0.52
SINH(-0.5 ) = -0.52
SINH( 1.0 ) = 1.17
SINH(-1.0 ) = -1.17
```

Related Topics

Expressions	Interpretations
Floating Point Constants	COSH Function
Integer Constants	TANH Function
Patterns	

Slots

Definition

A *slot* is the constructed unit in the Rules Element which stores a data value for objects or classes. It is the fundamental unit upon which rules and methods act to evaluate conditions or perform actions. The usage of slots depends upon whether the slot is defined as public or private.

Structure

For each property associated with a particular object or class name, the Rules Element constructs a slot. This construction is denoted by appending the name of the property to the name of the object or class, separated by a period (.):

```
object_name.property_name or class_name.property_name
```

Because the property associated with the object or class defines the data type, the slot data value belongs to one of the six elementary data type (integer, floating point, boolean, string, date, or time). If data encapsulation is required, the slot can be defined to be private to the object. Unless specified, slots are created as public and data protection is not provided. Property names associated with a particular object or class must be unique whether the slot is defined as public or private.

Scope

Unlike properties, the slot is local to a particular object or class. Initially all slots, regardless of its data type, have the value `UNKNOWN`. During knowledge processing the Rules Element tries to determine the data value of a slot when it is needed to evaluate a rule or method condition. In most cases the slot will be public. If data protection is desired, the application developer may decide to use a private slot to store the data value. While public slot values are accessible globally by any rule or method, private slot values are accessible only locally through a method associated with the class, object, or property named by the slot.

Private slots, together with methods, let application developers enforce object encapsulation when particular functionality and objects should be hidden. The developer can be sure that no part of their application will modify the stored value other than the object's associated method. On the other hand, a private slot value set by the action of a method has no consequence on rule processing. The resulting data value will not produce forward chaining of any kind (either through semantic gates or forward action effects) because the private slot cannot be used in rules. To ensure data protection is maintained for private slots the following behavior is enforced:

- Private slots cannot appear in the conditions and actions or rules.
- Private slots are ignored in pattern matching conditions.
- Methods attached to a private slot can only be triggered from another method.
- Interpretations on a private slot are only valid in the method associated with the object, class, or property named by the slot and the `SELF` keyword must be used.

Public slots, in contrast to private slots, are the fundamental unit upon which rule test conditions act. The Rules Element tries to determine a value for a public slot through a variety of means defined by the system default Order of Sources strategy. Also, public slots can be used without restriction in method conditions and actions where they can have consequences on rule processing (through semantic gates and forward action affects).

Private and public slots have the same value and property inheritance behavior. It is legal to inherit up and down from a private slot. It is also legal to inherit into a private slot. Inheritance of slots is controlled by inheritability strategies.

Creation

Slots can be created in either of two ways:

- Implicitly, by using a previously undefined slot name in a condition or action in the Rule, Object, Class or Method editor.
- Interactively, by adding previously created property names to the list that appears in the Class or Object editor.

By default slots are created as globally accessible (public) and data protection is not provided. A private slot is created by setting the Private attribute in the Meta-Slot editor. Slots created from properties inherited from a parent class or object which are private will also be private in the child.

Deletion

Slots are destroyed by removing property names from the Class or Object editor list of associated properties.

Access

How a slot's current value is obtained depends upon whether the slot is private or public.

A private slot's data value is always obtained by a method associated with the slot (or its class, object, or property). The method is triggered through the use of the `SendMessage` operator in the conditions or actions of another method. The method used to determine a private slot's value can never be triggered from a rule since it is not legal to specify private slot names in rule conditions and actions. Also, the private slot name cannot appear directly in the conditions or actions of a method. It is only legal to refer to the private slot name using the `SELF` operator in the conditions or actions of the method associated with the slot. The construction `SELF.property_name` allows an inherited method to properly access a private slot used in the method's conditions or actions. Use of the actual private slot name is not legal even in the method and will produce an error message during compilation.

A public slot's data value is obtained by the standard Order of Sources method defined by the Rules Element (see Order of Sources Method).

Related Topics

Patterns	Methods
Forward Chaining	Meta-Slots
Interpretations	SELF
Inference	SendMessage Operator
Inheritance	If Change Method
Rules	Order of Sources Method

SQRT Function

Definition

The *SQRT function* is used in expressions to find the square root of a floating point number. The expression can appear on the left-hand side or right-hand side of rules and methods.

Syntax

The function consists of the word `SQRT` followed by a single argument in parentheses:

```
SQRT(x)
```

Argument

The argument may be any expression yielding a numerical result greater than or equal to 0.0. The expression may include patterns or interpretations.

If the value of the argument expression is an integer, it will be converted to an equivalent floating point value.

Result

The function returns a floating point result equal to the square root of the argument.

If the argument expression does not produce a numerical value, an error message is posted and the function result is NOTKNOWN.

Examples

The following examples illustrate the results of the SQRT function:

```
SQRT(0.0) = 0.0
SQRT(0.5) = 0.71
SQRT(1.0) = 1.0
SQRT(2)   = 1.41
SQRT(2.0) = 1.41
SQRT(4)   = 2.0
```

Related Topics

Expressions

Floating Point Constants

Integer Constants

Patterns

Interpretations

STDEV Function

Definition

The *STDEV function* is used in expressions to find the standard deviation of a set of numerical values. The expression can appear on the left-hand side or right-hand side of rules and methods.

Syntax

The function consists of the word `STDEV` followed by any number of arguments in parentheses:

```
STDEV(x1, x2, . . . , xn)
```

Arguments

Each argument may be any expression yielding a numerical or time-valued result. There may be either a list of arguments or a pattern matching list.

If some of the argument values are integers and some floating point, the integers will be converted to equivalent floating point values before computation.

Result

The function returns a floating point result equal to the standard deviation of all the argument values (the square root of the sum of the squares of the differences of the values from the mean divided by the number of values). For arguments that include patterns, it uses all values in the corresponding list.

If any argument is of a non-numeric type, an error message is posted and the function result is NOTKNOWN.

Examples

The following examples illustrate the results of the STDEV function:

```
STDEV(365,240,577)           = 139.09
STDEV(98.6,37.0,-273.18)    = 162.69
STDEV(12,11.7)              = 0.15
STDEV(TIME(8,4,23),TIME(3,6,11)) = NOTKNOWN
STDEV(123,"456")            = NOTKNOWN
```

If class Tank has four instances with capacity values of 6.3, 14.5, 12.9, and 9.0, then

```
STDEV(<Tank>.capacity) = 3.22
```

Related Topics

Expressions

Data Types

Patterns

Interpretations

AVERAGE Function

VAR Function

Strategy

Definition

Strategy options determine various aspects of the Rules Element's behavior under the control of the application developer or of the inference process itself.

Variations

Strategy options include three general varieties:

- Inference strategy controls the operation of the Rules Element's inference processing and the propagation of results from one inference rule or method to another rule.
- Inheritability strategy controls the inheritability of properties and their values from one object or class to another.
- Inheritance strategy controls the order in which an object's classes and parent objects are searched for the inherited values of its properties. If the same property can be inherited from more than one source, the strategy determines which source will actually be used.

See the sections "Inference Strategy," "Inheritability Strategy," and "Inheritance Strategy" for further information.

Control

Strategy options can be set either interactively, with the Strategy Monitor window (from the Expert menu), or dynamically in the course of inference processing itself, via the *Strategy* operator in the conditions or actions of a rule or method.

Related Topics

Rules	Inference Strategy
Methods	Inheritance Strategy
Actions	Inheritability Strategy
Inference	Strategy Operator
Inheritance	

Strategy Operator

Definition

The *strategy operator* is used in the conditions or actions of a rule or method to control or modify the system's global strategy settings.

Operands

The Strategy operator takes a single operand, which consists of a series of individual strategy options of the forms

```
@option=TRUE;
@option=FALSE;
```

Notice that the closing semicolon (;) is required, even for the last option in the list.

Parameters

The following strategy options are recognized:

Inference

@PWTRUE	Forward confirmed hypotheses (“Propagate when TRUE”).
@PWFALSE	Forward rejected hypotheses (“Propagate when FALSE”).
@PWNOTKNOWN	Forward NOTKNOWN hypotheses (“Propagate when NOTKNOWN”).
@PFACTIONS	Forward Action-Effects for rules (“Propagate forward actions”). Specifically controls the rule left-hand side and right-hand side Then part.
@PFEACTIONS	Forward Action-Effects for rules (“Propagate forward actions”). Controls only the rule right-hand side Else part.
@PFMACTIONS	Forward Action-Effects for methods (“Propagate forward actions”). Specifically controls the method left-hand side and right-hand side Then part.
@PFMEACTIONS	Forward Action-Effects for methods (“Propagate forward actions”). Controls only the method right-hand side Else part.
@PTGATES	Forward through gates (“Propagate through gates”).

@EXHBWRD	Exhaustive evaluation (“Exhaustive backward”).
@SOURCESON	Automatically trigger Order of Sources methods when value is needed.
@CACTIONSON	Automatically trigger If Change methods when value changes.
@VALIDUSER	Enable validation of input solicited from the user before input is accepted for inferencing.
@VALIDENGINE	Enable validation of input given by the system before input is accepted for inferencing (for example, from an Assign, Execute, or Retrieve).

Inheritability

@INHCLASSDOWN	Inherit class properties downward
@INHCLASSUP	Inherit class properties upward
@INHOBJDOWN	Inherit object properties downward
@INHOBJUP	Inherit object properties upward
@INHVALDOWN	Inherit property values downward
@INHVALUP	Inherit property values upward

Inheritance

@INHPARENT	Inherit object-first
@INHBREADTH	Inherit breadth-first

See the Inference Strategy, Inheritance Strategy, and Inheritability Strategy topics for further details on the meanings and effects of individual options.

Strategy Arguments Dialog

When entering a Strategy action in the Rule editor or Method editor, clicking in the space for the first operand displays an arguments dialog box for specifying the strategy arguments interactively, rather than by explicitly typing in the keywords listed above. The inference strategies shown in the dialog box have the following options that you can select:

ON	Enables the strategy until the next local strategy changes the setting.
OFF	Disables the strategy until the next local strategy changes the setting.
CURRENT	Invokes the corresponding Strategy Monitor window setting (on the Expert menu) until the next local strategy changes the setting.
GLOBAL	This option is used to synchronize control of the individual <u>Forward Action Effects</u> strategies (@PFEACTIONS, @PFMACTIONS, and @PFMECTIONS) with the setting of “Rule Global Forward Action-Effects” (@PFACTIONS) that appears in the Strategy Monitor window. For instance, you can selectively enable or disable Else actions from a rule, or you can select the GLOBAL option so the strategy behaves exactly as the rule Then actions setting.

In addition to the local strategy options described here, the strategic behavior of individual rules can be controlled by using certain special values for their inference priorities. See the Inference Priority Attribute topic for details.

Effect

The designated global strategy settings are enabled or disabled, as specified. Options not explicitly modified by changing the setting `CURRENT`, remain unchanged from their previous global settings.

Examples

The following are examples of actions using the `Strategy` operator:

```
Strategy @PWTRUE=TRUE ;@PWFALSE=TRUE ;@PWNOTKNOWN=FALSE ;
Strategy @INHPARENT=FALSE ;@INHBREADTH=TRUE ;
Strategy @INHOBBDOWN=TRUE ;
```

Related Topics

Rules

Methods

Actions

Inference

Inheritance

Inference Strategy

Inheritance Strategy

Inheritability Strategy

`NXP_Strategy` call from API

STRCAT Function

Definition

The *STRCAT function* is used in expressions to concatenate two character strings. The expression can appear on the left-hand side or right-hand side of rules and methods.

Syntax

The function consists of the word `STRCAT` followed by two arguments in parentheses:

```
STRCAT ( s1 , s2 )
```

Arguments

Each argument may be any expression yielding a string result. The expressions may include patterns or interpretations.

Result

The function returns a string result equal to the concatenation of the two argument strings.

If either argument expression does not produce a string value, an error message is posted and the function result is `NOTKNOWN`.

Examples

The following examples illustrate the results of the STRCAT function:

```
STRCAT("flap","doodle") = "flapdoodle"
STRCAT("flap","")      = "flap"
STRCAT("", "doodle")   = "doodle"
STRCAT("", "")         = ""
STRCAT("flap",s)       = "flapdoodle" if s="doodle"
STRCAT("red_",STRCAT("flap","doodle")) = "red_flapdoodle"
```

Related Topics

Expressions	STRLEN Function
String Constants	SUBSTRING Function
Patterns	STRFIND Function
Interpretations	STRUPPER Function
STRLOWER Function	

STRFIND Function

Definition

The *STRFIND function* is used in expressions to search a character string for another character string. The expression can appear on the left-hand side or right-hand side of rules and methods.

Syntax

The function consists of the word STRFIND followed by two arguments in parentheses:

```
STRFIND(s1, s2)
```

Arguments

Each argument may be any expression yielding a string result:

- The first argument (s1) is the string to be searched.
- The second argument (s2) specifies the string to search for.

The argument expressions may include patterns or interpretations.

Result

The function returns an integer result equal to the offset from the beginning of the first argument string (s1) to the first occurrence of the second string (s2). The search is case sensitive, therefore corresponding uppercase and lowercase letters (such as A and a) are considered different for purposes of the search. An offset of 0 denotes the first character in string s1 (no offset at all from the start of the string). If s1 does not contain s2, the function result is -1. If either argument expression does not produce a string value, an error message is posted and the function result is NOTKNOWN.

Examples

The following examples illustrate the results of the STRFIND function:

```
STRFIND("SHAZAM!","SHA") = 0
STRFIND("SHAZAM!","A") = 2
STRFIND("SHAZAM!","ZAM") = 3
STRFIND("SHAZAM!","ZAMS") = -1
STRFIND("SHAZAM!","ZaM") = -1
STRFIND("SHAZAM!","") = 0
STRFIND("","SHAZAM!") = -1
```

Related Topics

Expressions	SUBSTRING Function	STRUPPER
Function		
String Constants	STRLEN Function	STRLOWER Function
Patterns	STRCAT Function	Interpretations
CHARFIND Function		

String Constants

Definition

A string constant is a sequence of text characters used directly as a data value in a Rules Element rule or method, or as a property of an object.

Syntax

A string constant consists of any sequence of characters enclosed in double quotation marks (" . . . "). To include the double quote character itself within a string, precede it with a backslash (\). The backslash is merely a syntactic marker, and will not be included in the string; any backslash not followed immediately by a quote character is considered to stand for itself and will be included in the string.

Note: in many places where arguments must be string constants, you can include an interpreted slot with the syntax @v(slot).

Examples

The following are valid string constants:

```
"e"
"SHAZAM!"
"Jack and Jill went up the hill"
"Press \"Return\" to continue"
"Either\Or"
"%?!*!"
"1789"
""
```

The last example denotes the empty string, which contains no characters at all. Notice that the string 1789 is merely a sequence of characters, and is not the same as the integer 1789.

The following are not valid string constants:

SHAZAM!	Not enclosed in quotes.
"Either\Or	Quotes not balanced.
"	Quotes not balanced.
"Press "Return" to continue"	Quotes not backslashed.

Related Topics

Objects	STRLEN Function
Properties	STRCAT Function
Rules	SUBSTRING Function
Data Types	STRFIND Function
Integer Constants	STRUPPER Function
String Formats	STRLOWER Function

String Formats

Definition

A *string format* specifies the representation of a string value for input and output purposes.

Syntax

This section defines the syntax of format elements for string-valued properties only. See the section titled “Formats” for the syntax of formats in general.

The following special character is meaningful in string formats:

`s` Placeholder for value of string

Like all formats, those for string values may include strings of literal characters enclosed in double quotation marks (" . . . "), and may also include the wild-card character (*). Format elements beginning with an exclamation point (!) are ignored in database transactions; they are meaningful only for direct interaction with the user via the screen and keyboard.

Input

On input, each element in the format list is tried in order until one of them matches the input text. If no match is found, the input is rejected and an error message is displayed on the screen. The following conventions apply:

- Strings of literal characters enclosed in double quotation marks must match exactly, except that no distinction is made between uppercase and lowercase letters.
- The wild-card character (*) matches any sequence of zero or more characters.
- The letter `s` in the format specification also matches any sequence of zero or more characters, and in addition assigns these characters as the value of the string slot being read.

Output

On output, only the first element in the format list is used (except if preceded by an !):

- Strings of literal characters enclosed in double quotation marks are reproduced exactly in the output.
- The letter `s` in the format specification is replaced in the output by the value of the string slot being written.
- The wild-card character (`*`) is ignored on output.

Default

The default system format for strings is defined in the `ckbres.format` module in the file `nrxrun.dat`. The standard default format is simply:

`s`

If necessary, the `ckbres.format` module in the file `nrxrun.dat` can be modified to substitute another default format instead.

Example

The following example illustrates the use of string formats:

Example 1 Format: `"Color is "s;s;@N="Color is undefined"`

Value	Output	Comments
<code>"red"</code>	<code>Color is red</code>	Uses first element
<code>NOTKNOWN</code>	<code>Color is undefined</code>	Uses last (@N=) element

Input	Value	Comments
<code>Color is blue</code>	<code>"blue"</code>	Matches first element
<code>Color Is Blue</code>	<code>"Blue"</code>	Match is case-insensitive
<code>green</code>	<code>"green"</code>	Matches second element
<code>NOTKNOWN</code>	<code>NOTKNOWN</code>	Reserved word
<code>Color is undefined</code>	<code>NOTKNOWN</code>	Matches last (@N=) element
<code>undefined</code>	<code>"undefined"</code>	Matches second element

Example 2 Format: `!"Color is "s;s`

Value	Output on Screen	Output in Database
<code>"red"</code>	<code>Color is red</code>	<code>red</code>

Example 3 Format: `*" is "s;s`

Value	Input	Comments
<code>red</code>	<code>"The color of this car is red"</code>	<code>"The color of this car" is matched by *</code>

Related Topics

Formats

Format Attribute

String Constants

STRLEN Function

Definition

The *STRLEN function* is used in expressions to find the length of a character string. The expression can appear on the left-hand side or right-hand side of rules and methods.

Syntax

The function consists of the word `STRLEN` followed by a single argument in parentheses:

```
STRLEN( s )
```

Argument

The argument may be any expression yielding a string result. The expression may include patterns or interpretations.

Result

The function returns an integer result equal to the number of characters in the argument string.

If the argument expression do not produce a string value, an error message is posted and the function result is `NOTKNOWN`.

Examples

The following examples illustrate the results of the `STRLEN` function:

```
STRLEN( "a" )      = 1
STRLEN( "SHAZAM!" ) = 7
STRLEN( "1492" )  = 4
STRLEN( " " )     = 0
```

Related Topics

Expressions

String Constants

Patterns

Interpretations

`STRLOWER` Function

`STRCAT` Function

`SUBSTRING` Function

`STRFIND` Function

`STRUPPER` Function

STRLOWER Function

Definition

The *STRLOWER function* is used in expressions to convert a character string to lowercase. The expression can appear on the left-hand side or right-hand side of rules and methods.

Syntax

The function consists of the word `STRLOWER` followed by a single argument in parentheses:

```
STRLOWER( s )
```


Argument

The argument may be any expression yielding a string result. The expression may include patterns or interpretations.

Result

The function returns a string result equivalent to the argument string with all letters converted to lowercase. Nonalphabetic characters are unaffected.

If the argument expression does not produce a string value, an error message is posted and the function result is NOTKNOWN.

Examples

The following examples illustrate the results of the STRLOWER function:

```
STRLOWER ( " SHAZAM! " ) = "shazam! "
STRLOWER ( " ShaZam! " ) = "shazam! "
STRLOWER ( " shazam! " ) = "shazam! "
STRLOWER ( " 23 SKIDOO " ) = "23 ski000"
STRLOWER ( " " ) = " "
```

Related Topics

Expressions	STRUPPER Function
String Constants	STRLEN Function
Patterns	STRCAT Function
Interpretations	SUBSTRING Function
STRFIND Function	

STRUPPER Function

Definition

The *STRUPPER function* is used in expressions to convert a character string to uppercase. The expression can appear on the left-hand side or right-hand side of rules and methods.

Syntax

The function consists of the word STRUPPER followed by a single argument in parentheses:

```
STRUPPER ( s )
```

Argument

The argument may be any expression yielding a string result. The expression may include patterns or interpretations.

Result

The function returns a string result equivalent to the argument string with all letters converted to uppercase. Nonalphabetic characters are unaffected.

If the argument expression does not produce a string value, an error message is posted and the function result is NOTKNOWN.

Examples

The following examples illustrate the results of the `STRUPPER` function:

```
STRUPPER("shazam!") = "SHAZAM!"
STRUPPER("ShaZam!") = "SHAZAM!"
STRUPPER("SHAZAM!") = "SHAZAM!"
STRUPPER("23 skidoo") = "23 SKIDOO"
STRUPPER("") = ""
```

Related Topics

Expressions	<code>STRLOWER</code> Function
String Constants	<code>STRLEN</code> Function
Patterns	<code>STRCAT</code> Function
Interpretations	<code>SUBSTRING</code> Function
<code>STRFIND</code> Function	

STR2BOOL Function

Definition

The *STR2BOOL* function is used in expressions to convert a character string to the boolean value it represents. The expression can appear on the left-hand side or right-hand side of rules and methods.

Syntax

The function consists of the word `STR2BOOL` followed by one or two arguments in parentheses:

```
STR2BOOL(s)
STR2BOOL(s, f)
```

Argument

Each argument may be any expression yielding a string result:

- The first argument (`s`) is the string to be converted.
- The optional second argument (`f`) is a string specifying the format by which the first argument is to be interpreted. See “Boolean Formats” for the syntax and meaning of this string.

The argument expressions may include patterns or interpretations.

Result

The function returns a boolean result equal to the boolean value represented by string `s`, interpreted according to format `f`. If no format argument is given, the default system format for booleans (defined in the `ckbres.format` module in the file `nrxrun.dat`) is used.

If the string `s` cannot be interpreted as a boolean value under the given format, the function result is `NOTKNOWN`.

Examples

The following examples illustrate the results of the STR2BOOL function:

```
STR2BOOL( "FALSE" )           = FALSE
STR2BOOL( "Nope" , "Yup;Nope" ) = FALSE
STR2BOOL( "FALSE" , "Yup;Nope" ) = NOTKNOWN
STR2BOOL( "MAYBE" )          = NOTKNOWN
STR2BOOL( " " )              = NOTKNOWN
```

Related Topics

Expressions

String Constants

Boolean Constants

Boolean Formats

Patterns

Interpretations

BOOL2STR Function

STR2DATE Function

Definition

The *STR2DATE* function is used in expressions to convert a character string to the date value it represents. The expression can appear on the left-hand side or right-hand side of rules and methods.

Syntax

The function consists of the word STR2DATE followed by one or two arguments in parentheses:

```
STR2DATE( s )
STR2DATE( s , f )
```

Argument

Each argument may be any expression yielding a string result:

- The first argument (s) is the string to be converted.
- The optional second argument (f) is a string specifying the format by which the first argument is to be interpreted. See “Date Formats” for the syntax and meaning of this string.

The argument expressions may include patterns or interpretations.

Result

The function returns a date result equal to the date represented by string s, interpreted according to formatf. If no format argument is given, the default system format for dates (defined in the `ckbres.format` module in the file `nxrun.dat`) is used.

If the string s cannot be interpreted as a date under the given format, the function result is NOTKNOWN.

Examples

The following examples illustrate the results of the `STR2DATE` function:

```
STR2DATE("jun 16 1904")      = DATE(1904,6,16)
STR2DATE("6/16/04", "m/d/YY") = DATE(1904,6,16)
STR2DATE("Bloomsday")      = NOTKNOWN
STR2DATE(" ")              = NOTKNOWN
```

Related Topics

Expressions

String Constants

DATE Function

TIME Function

Date Formats

Patterns

Interpretations

DATE2STR Function

STR2FLOAT Function

Definition

The `STR2FLOAT` function is used in expressions to convert a character string to the floating point value it represents. The expression can appear on the left-hand side or right-hand side of rules and methods.

Syntax

The function consists of the word `STR2FLOAT` followed by one or two arguments in parentheses:

```
STR2FLOAT(s)
STR2FLOAT(s, f)
```

Argument

Each argument may be any expression yielding a string result:

- The first argument (`s`) is the string to be converted.
- The optional second argument (`f`) is a string specifying the format by which the first argument is to be interpreted. See “Floating Point Formats” for the syntax and meaning of this string.

The argument expressions may include patterns or interpretations.

Result

The function returns a floating point result equal to the numeric value represented by string `s`, interpreted according to format `f`. If no format argument is given, the default system format for floating point numbers (defined in the `ckbres.format` module in the file `nrxrun.dat`) is used.

If the string `s` cannot be interpreted as a floating point value under the given format, the function result is `NOTKNOWN`.

Examples

The following examples illustrate the results of the `STR2FLOAT` function:

```
STR2FLOAT("98.6")          = 98.6
STR2FLOAT("-273.18")       = -273.18
STR2FLOAT("98.6 degrees", "0.0*") = 98.6
STR2FLOAT("1,234.5", "k,u.0") = 1234.5
```

```
STR2FLOAT("degrees", "0.0*") = NOTKNOWN
STR2FLOAT(" ") = NOTKNOWN
```

Related Topics

Expressions

String Constants

Floating Point Constants

Floating Point Formats

Patterns

Interpretations

FLOAT2STR Function

STR2INT Function

Definition

The `STR2INT` function is used in expressions to convert a character string to the integer value it represents. The expression can appear on the left-hand side or right-hand side of rules and methods.

Syntax

The function consists of the word `STR2INT` followed by one or two arguments in parentheses:

```
STR2INT(s)
STR2INT(s, f)
```

Argument

Each argument may be any expression yielding a string result:

- The first argument (`s`) is the string to be converted.
- The optional second argument (`f`) is a string specifying the format by which the first argument is to be interpreted. See “Integer Formats” for the syntax and meaning of this string.

The argument expressions may include patterns or interpretations.

Result

The function returns an integer result equal to the numeric value represented by string `s`, interpreted according to format `f`. If no format argument is given, the default system format for integers (defined in the `ckbres.format` module in the file `nrxrun.dat`) is used.

If the string `s` cannot be interpreted as an integer value under the given format, the function result is `NOTKNOWN`.

Examples

The following examples illustrate the results of the `STR2INT` function:

```
STR2INT("23") = 23
STR2INT("23 skidoo", "d*") = 23
STR2INT("4F", "x") = 79
STR2INT("skidoo", "d*") = NOTKNOWN
STR2INT(" ") = NOTKNOWN
```

Related Topics

Expressions
String Constants
Integer Constants
Integer Formats

Patterns
Interpretations
INT2STR Function

STR2TIME Function

Definition

The `STR2TIME` function is used in expressions to convert a character string to the time value it represents. The expression can appear on the left-hand side or right-hand side of rules and methods.

Syntax

The function consists of the word `STR2TIME` followed by one or two arguments in parentheses:

```
STR2TIME ( s )
STR2TIME ( s , f )
```

Argument

Each argument may be any expression yielding a string result:

- The first argument (`s`) is the string to be converted.
- The optional second argument (`f`) is a string specifying the format by which the first argument is to be interpreted. See “Time Formats” for the syntax and meaning of this string.

The argument expressions may include patterns or interpretations.

Result

The function returns a time result equal to the time interval represented by string `s`, interpreted according to format `f`. If no format argument is given, the default system format for times (defined in the `ckbres.format` module in the file `nrxrun.dat`) is used.

If the string `s` cannot be interpreted as a time under the given format, the function result is `NOTKNOWN`.

Examples

The following examples illustrate the results of the `STR2TIME` function:

```
STR2TIME("0 years 29 days 12:44:03") = TIME(0,0,29,12,44,3)
STR2TIME("29 days 12:44:03") = NOTKNOWN
STR2TIME("29 days 12:44:03", "dd*hh:mm:ss") =
TIME(0,0,29,12,44,3)
STR2TIME("12:44:03") = TIME(0,0,0,12,44,3)
STR2TIME("") = NOTKNOWN
```

Related Topics

Expressions
String Constants
DATE Function
TIME Function

Time Formats
Patterns
Interpretations
TIME2STR Function

SUBSTRING Function

Definition

The *SUBSTRING function* is used in expressions to extract a substring of a given character string. The expression can appear on the left-hand side or right-hand side of rules and methods.

Syntax

The function consists of the word `SUBSTRING` followed by three arguments in parentheses:

```
SUBSTRING ( s , m , n )
```

Arguments

Each argument may be any expression yielding a result of the appropriate type:

- The first argument (*s*) is the string from which the substring is to be extracted.
- The second argument (*m*) is an integer giving the offset in characters from the beginning of the string to the beginning of the substring.
- The third argument (*n*) is an integer giving the length of the substring in characters.

The second and third arguments may be given as floating point values, which will be converted to equivalent integers. The argument expressions may include patterns or interpretations.

Result

The function returns the substring of *n* characters taken from string *s* beginning at offset *m*.

An offset of 0 denotes the first character in string *s* (no offset at all from the start of the string). If the end of the string is encountered prematurely, the resulting substring will be shorter than the requested length *n*. If the offset *m* lies beyond the end of string *s*, the function will return the empty string.

If any of the argument expressions does not produce a value of the appropriate type, an error message is posted and the function result is `NOTKNOWN`.

Examples

The following examples illustrate the results of the SUBSTRING function:

```
SUBSTRING("SHAZAM!", 0, 2) = "SH"
SUBSTRING("SHAZAM!", 3, 3) = "ZAM"
SUBSTRING("SHAZAM!", 3, 10) = "ZAM!"
SUBSTRING("SHAZAM!", 0, 7) = "SHAZAM!"
SUBSTRING("SHAZAM!", 0, 10) = "SHAZAM!"
SUBSTRING("SHAZAM!", 3, 0) = ""
SUBSTRING("SHAZAM!", 10, 3) = ""
SUBSTRING("SHAZAM!", -3, 2) = ""
SUBSTRING("SHAZAM!", -3, 5) = "SH"
SUBSTRING("", 0, 3) = ""
```

Related Topics

Expressions	SUBSTRING Function
String Constants	STRLEN Function
Patterns	STRCAT Function
Interpretations	STRUPPER Function
STRLOWER Function	

SUM Function

Definition

The *SUM function* is used in expressions to find the sum of a set of numerical values. The expression can appear on the left-hand side or right-hand side of rules and methods.

Syntax

The function consists of the word `SUM` followed by any number of arguments in parentheses:

```
SUM(x1, x2, . . . , xn)
```

Arguments

Each argument may be any expression yielding a numerical or time-valued result. There may be either a list of arguments or a pattern matching list.

If some of the argument values are integers and some floating point, the integers will be converted to equivalent floating point values before computation.

Result

The function adds together all the argument values and returns their sum. For arguments that include patterns, it adds all values in the corresponding list.

Integer and floating point values may be mixed in the same sum, but time values can be added only to each other. If numeric and time arguments are mixed, or if any argument is of another type, an error message is posted and the function result is `NOTKNOWN`.

Examples

The following examples illustrate the results of the SUM function:

```
SUM(365,240,577)           = 1182
SUM(98.6,37.0,-273.18)    = -137.58
SUM(12,11.7)              = 23.7
SUM(TIME(8,4,23),TIME(3,6,11)) = TIME(11,10,34)
SUM(123,"456")           = NOTKNOWN
```

If class Tank has four instances with capacity values of 6.3, 14.5, 12.9, and 9.0, then

```
SUM(<Tank>.capacity) = 42.7
```

Related Topics

Expressions

Data Types

DATE Function

TIME Function

Patterns

Interpretations

PROD Function

TAN Function

Definition

The *TAN function* is used in expressions to find the tangent of a floating point number. The expression can appear on the left-hand side or right-hand side of rules and methods.

Syntax

The function consists of the word TAN followed by a single argument in parentheses:

```
TAN(x)
```

Argument

The argument may be any expression yielding a numerical result. The expression may include patterns or interpretations.

If the value of the argument expression is an integer, it will be converted to an equivalent floating point value.

Result

The function returns a floating point result equal to the tangent of the argument. The argument is assumed to be expressed in radians.

If the argument expression does not produce a numerical value, an error message is posted and the function result is NOTKNOWN.

Examples

The following examples illustrate the results of the TAN function:

```
TAN(0.0)           = 0.0
TAN(3.14 / 4)      = 1.0
TAN(3.14 / 3)      = 1.73
TAN(3.14)          = 0.0
TAN(-3.14 / 3)     = -1.73
```

Related Topics

Expressions	Interpretations
Floating Point Constants	SIN Function
Integer Constants	COS Function
Patterns	ATAN Function

TANH Function

Definition

The *TANH function* is used in expressions to find the hyperbolic tangent of a floating point number. The expression can appear on the left-hand side or right-hand side of rules and methods.

Syntax

The function consists of the word `TANH` followed by a single argument in parentheses:

```
TANH ( x )
```

Argument

The argument may be any expression yielding a numerical result. The expression may include patterns or interpretations.

If the value of the argument expression is an integer, it will be converted to an equivalent floating point value.

Result

The function returns a floating point result equal to the hyperbolic tangent of the argument.

If the argument expression does not produce a numerical value, an error message is posted and the function result is `NOTKNOWN`.

Examples

The following examples illustrate the results of the `TANH` function:

```
TANH ( 0.0 ) = 0.0  
TANH ( 0.5 ) = 0.46  
TANH ( -0.5 ) = -0.46  
TANH ( 1.0 ) = 0.76  
TANH ( -1.0 ) = -0.76
```

Related Topics

Expressions	Interpretations
Floating Point Constants	SINH Function
Integer Constants	COSH Function
Patterns	

Time Formats

Definition

A time format specifies the representation of a time value in text form for input and output purposes.

Syntax

This section defines the syntax of format elements for times only. See the section titled “Formats” for the syntax of formats in general.

The following special characters are meaningful in time formats:

Y,y	Years field
M,m	Months or minutes field
D,d	Days field
H,h	Hours field
S,s	Seconds field

Time formats are case insensitive. A series of Ms or ms immediately preceded by an hours field denotes a minutes field; otherwise it is interpreted as months instead.

Like all formats, those for times may include strings of literal characters enclosed in double quotation marks (" . . . "), and may also include the wild-card character (*). Format elements beginning with an exclamation point (!) are ignored in database transactions; they are meaningful only for direct interaction with the user via the screen and keyboard.

Input

On input, each element in the format list is tried in order until one of them matches the input text. If no match is found, the input is rejected and an error message is displayed on the screen. The following conventions apply:

- Input values of any length are recognized; the number of letters used to specify a field in the format is ignored.
- Strings of literal characters enclosed in double quotation marks must match exactly, except that no distinction is made between uppercase and lowercase letters.
- The wild-card character (*) matches any sequence of zero or more characters.

Output

On output, only the first element in the format list is used:

- Strings of literal characters enclosed in double quotation marks are reproduced exactly in the output.
- The wild-card character (*) is ignored on output.
- The number of letters used to define a field within a format element specifies the minimum number of digits to be used in that field’s output representation. Values shorter than this will be padded with leading zeros; longer values will be represented in full, using more than the specified number of digits.

Example

The format

```
hh:mm:ss;*h*m*s*
```

will format times on output in the form

```
02:06:50
```

and will accept them on input in such forms as

```
02:06:50
```

```
2:06:50
```

```
2:6:50
```

(matching the first format element) or

The elapsed time is 2 hours, 6 minutes, and 50 seconds.

(matching the second).

Default

The default system format for times is defined in the `ckbres.format` module in the file `nrxrun.dat`. The standard default format is

```
y" years "d" days "hh:mm:ss;yy dd hh:mm:ss;hh:mm:ss
```

This format will output times in the form

```
3 years 193 days 22:34:17
```

and will accept them as input in any of the forms

```
3 years 193 days 22:34:17
```

```
3 193 22:34:17
```

```
22:34:17
```

If necessary, the `ckbres.format` module in the file `nrxrun.dat` can be modified to substitute another default format instead.

Related Topics

Formats

Format Attribute

DATE Function

TIME Function

Date Formats

TIME Function

Definition

A time is a data value representing an interval of duration or elapsed time. See also the DATE Function topic.

Time Syntax

A time constant can be specified in either of two formats, similar to those for dates (see the DATE Function topic):

```
TIME(hours, minutes, seconds)
```

```
TIME(years, months, days, hours, minutes, seconds)
```

In this case, however, the ranges of the parameters are different:

0	≤	years	≤	32767
0	≤	months	≤	255
0	≤	days	≤	32767
0	≤	hours	≤	255
0	≤	minutes	≤	255
0	≤	seconds	≤	255

For example,

`TIME(8,4,23)`

denotes a time interval of 8 hours, 4 minutes, and 23 seconds, while

`TIME(3,6,11,22,34,17)`

denotes an interval of 3 years, 6 months, 11 days, 22 hours, 34 minutes, and 17 seconds.

Expressions

Dates and times can be combined arithmetically in various ways. You can add or subtract two time intervals to produce a third interval representing their sum or difference, subtract two dates to find the interval between them, or add or subtract a date and a time to produce another date. You can also multiply or divide a time by a number (integer or floating point). In summary, here are the valid arithmetic operations on dates and times:

time + time	yields time
time - time	yields time
date - date	yields time
date + time	yields date
date - time	yields date
number * time	yields time
time * number	yields time
time / number	yields time

Note: When you edit a rule, the Time function converts the values of its data to the next possible higher unit. For example, if you add 32 days to a date by using `Time(0, 0, 32, 0, 0, 0) + date`, the Time function converts it to `Time(0, 1, 2, 0, 0, 0)`. These conversions are not always correct for conversions from days to months or years. Therefore, set the number of days to equal to or less than 30.

Related Topics

Date	HOUR Function
Data Types	MINUTE Function
Expressions	SECOND Function
Time Formats	NOW Function

TIME2FLOAT Function

Definition

The *TIME2FLOAT* function is used in expressions to convert a time to an equivalent floating point value. The expression can appear on the left-hand side or right-hand side of rules and methods.

Syntax

The function consists of the word `TIME2FLOAT` followed by a single argument in parentheses:

```
TIME2FLOAT ( t )
```

Argument

The argument may be any expression yielding a time result. The expression may include patterns or interpretations.

Result

The function returns a floating point result representing the number of seconds equivalent to the given time `t`.

Examples

The following examples illustrate the results of the `TIME2FLOAT` function:

```
TIME2FLOAT ( TIME ( 3 , 6 , 11 , 22 , 34 , 17 ) ) = 111515657.0
TIME2FLOAT ( TIME ( 8 , 4 , 23 ) )           = 29063.0
TIME2FLOAT ( " 8 : 4 : 23 " )                = NOTKNOWN
```

Related Topics

Expressions

[DATE Function](#)

[TIME Function](#)

[Patterns](#)

Interpretations

[FLOAT2TIME Function](#)

[DATE2FLOAT Function](#)

TIME2STR Function

Definition

The `TIME2STR` function is used in expressions to convert a time value to an equivalent character string. The expression can appear on the left-hand side or right-hand side of rules and methods.

Syntax

The function consists of the word `TIME2STR` followed by one or two arguments in parentheses:

```
TIME2STR ( t )
TIME2STR ( t , f )
```

Argument

Each argument may be any expression yielding a result of the appropriate type:

- The first argument (`t`) is the time to be converted.
- The optional second argument (`f`) is a string specifying the format under which the first argument is to be converted. See “Time Formats” for the syntax and meaning of this string.

The argument expressions may include patterns or interpretations.

Result

The function returns a string result representing the time value of argument `t`, converted according to format `f`. If no format argument is given, the default system format for times (defined in the `ckbres.format` module in the file `nrxrun.dat`) is used.

Examples

The following examples illustrate the results of the `TIME2STR` function:

```
TIME2STR(TIME(0,0,29,12,44,03)) = "0 years 29 days 12:44:03"
TIME2STR(TIME(0,0,29,12,44,03), "dd\" days \"hh:mm:ss") =
    "29 days 12:44:03"
TIME2STR(TIME(0,0,0,12,44,03)) = "12:44:03"
```

Related Topics

Expressions

String Constants

DATE Function

TIME Function

Time Formats

Patterns

Interpretations

STR2TIME Function

UnloadKB Operator

Definition

The *UnloadKB operator* is used in the conditions or actions of a rule or method to unload or disable a knowledge base.

Operands

The `UnloadKB` operator takes one or two operands:

- The first operand is a string constant or interpretation which evaluates to a string constant (using the `@V(object.prop)` syntax) specifying the name of the file containing the knowledge base to be unloaded. It must be between double quotes.
- The optional second operand specifies the knowledge base's load level, and must be one of the following:

```
@LEVEL=ENABLE ;
@LEVEL=DISABLEWEAK ;
@LEVEL=DISABLESTRONG ;
@LEVEL=DELETE ;
@LEVEL=WIPEOUT ;
```

(Note that the closing semicolon is required.) If the second operand is omitted, a load level of `DELETE` is assumed by default.

UnloadKB Dialog

When entering an `UnLoadKB` action in the Rule editor or Method editor, clicking in the space for the second operand displays a special dialog box for specifying the load level interactively, rather than by explicitly typing in the keywords listed above:



Effect

The knowledge base named as the first operand is unloaded from memory or changed to the load level specified by the second operand. Definitions not belonging to the given knowledge base remain in effect.

Load levels

The effects of the various load levels are as follows:

- ENABLE:** All definitions in the knowledge base are fully effective and operational, including objects, classes, properties, rules, and methods.
- DISABLEWEAK:** Object, class, and property definitions from the knowledge base remain in effect. Rules and methods remain defined, but become temporarily disabled and unavailable for inference processing; they can later be reenabled with `LoadKB`. Any disabled rules or methods already on the agenda remain there and will be processed normally.
- DISABLESTRONG:** Object, class, and property definitions from the knowledge base remain in effect. Rules and methods remain defined, but become temporarily disabled and unavailable for inference processing; they can later be reenabled with `LoadKB`. Any such disabled rules or methods already on the agenda are removed from the agenda and will not be processed.
- DELETE:** Object, class, and property definitions from the knowledge base remain in effect. Rules and methods are permanently deleted from memory.

and no longer available for inference processing; they can be reenabled only by reloading the knowledge base with LoadKB.

WIPEOUT:

All definitions from the knowledge base are permanently deleted from memory, including objects, classes, properties, rules, and methods; they can be reenabled only by reloading the knowledge base with LoadKB.

Examples

The following are examples of actions using the LoadKB operator:

```
UnloadKB "      Inventory.kb"
UnloadKB "      Inventory.kb"      @LEVEL=DISABLEWEAK
UnloadKB "      "Inventory.kb"      @LEVEL=DISABLESTRONG
UnloadKB "      Inventory.kb"      @LEVEL=DELETE
UnloadKB "      Inventory.kb"      @LEVEL=WIPEOUT
```

Related Topics

Rules

Methods

LoadKB Operator

Objects

ClassesString Constants

PropertiesActions

Agenda

NXP_UnloadKB function of API

Value Property

Definition

The special property named `Value` holds the data value (if any) associated directly with an object or class itself. Together the object and special property form a slot of any data type.

Type

The `Value` property is defined to be of type `Special`, allowing it to take on values of different data types for different objects. For any given object, however, its value is restricted to exactly one of the six elementary data types.

Access

The current value of an object's `Value` property is ordinarily denoted simply by the name of the object itself, with no qualifying property name. If `the_object` is the name of an object, the expressions

`the_object`

and

`the_object.Value`

are equivalent.

Restrictions

You cannot perform a pattern matching over a list of objects' `Value` property. The value property will never inherit a value nor a method.

Modifying

The `value` property associated with a particular object can be changed by assigning a new value directly to the name of the object itself in either of two ways:

- Explicitly, by executing the `Assign` operator in a condition or an action.
- Interactively, via the `Volunteer` command.

Related Topics

Objects

Properties

Data Types

Conditions

Actions

Assign Operator

VAR Function

Definition

The *VAR function* is used in expressions to find the variance of a set of numerical values. The expression can appear on the left-hand side or right-hand side of rules and methods.

Syntax

The function consists of the word `VAR` followed by any number of arguments in parentheses:

```
VAR(x1, x2, . . . , xn)
```

Arguments

Each argument may be any expression yielding a numerical result. The expressions may include existential patterns or interpretations; universal patterns are not allowed.

If some of the argument values are integers and some floating point, the integers will be converted to equivalent floating point values before computation.

Result

The function returns a floating point result equal to the statistical variance of all the argument values (the sum of the squares of the differences of the values from the mean divided by the number of values). For arguments that include patterns, it uses all values in the corresponding list.

If any argument is of a non-numeric type, an error message is posted and the function result is `NOTKNOWN`.

Examples

The following examples illustrate the results of the `VAR` function:

```
VAR(365, 240, 577)           = 19348.66
VAR(98.6, 37.0, -273.18)    = 26469.61
VAR(12, 11.7)               =      0.02
```

```
VAR(TIME(8,4,23),TIME(3,6,11)) = NOTKNOWN
VAR(123,"456") = NOTKNOWN
```

If class Tank has four instances with capacity values of 6.3, 14.5, 12.9, and 9.0, then

```
VAR(<Tank>.capacity) = 10.38
```

Related Topics

Expressions
Data Types
Patterns

Interpretations
AVERAGE Function
STDEV Function

WEEKDAY Function

Definition

The *WEEKDAY function* is used in expressions to find the day of the week corresponding to a given date. The expression can appear on the left-hand side or right-hand side of rules and methods.

Syntax

The function consists of the word `WEEKDAY` followed by a single argument in parentheses:

```
WEEKDAY(d)
```

Argument

The argument may be any expression yielding a date result. The expression may include patterns or interpretations.

Result

The function returns an integer result representing the day of the week corresponding to the given date argument. The result ranges from 1 (Monday) to 7 (Sunday).

If the argument expression does not produce a date value, an error message is posted and the function result is `NOTKNOWN`.

Examples

The following examples illustrate the results of the `WEEKDAY` function:

```
WEEKDAY( DATE(1492,10,12) ) = 4
WEEKDAY( DATE(1981,6,8,21,8,46) ) = 1
WEEKDAY( TIME(8,4,23) ) = NOTKNOWN
WEEKDAY( TIME(3,6,11,22,34,17) ) = NOTKNOWN
WEEKDAY( "October 12, 1492" ) = NOTKNOWN
```

Related Topics

Expressions
DATE Function
TIME Function
Patterns

YEAR Function
MONTH Function
DAY Function
HOUR Function

Interpretations	MINUTE Function
YEARDAY Function	SECOND Function
NOW Function	

Why Attribute

Definition

The *why attribute* associated with a slot, rule, or method specifies the text to be displayed on the screen when the end user requests an explanation for the system's current focus of attention.

Usage

The text of the why attribute is displayed in a dialog window whenever the end user selects the Why option from the session control panel of the Rules Element main window. The text describes the inferencing links leading to the slot displayed in the session control panel. The dialog window displays two buttons that let the end user traverse the backward chaining links starting from the current focus of attention:

- The Why button displays the why text associated with the next rule in the backward chaining links.
- The How button displays the why text associated with the previous rule in the backward chaining links.

Creation

The why text is specified or edited by typing into the box labeled why in the Rule editor, Method editor, or Meta-Slot editor. The supplied text has the following effect on the explanation dialog window:

Rule Editor	Why text appears in the bottom box that normally gives information about left-hand side conditions.
Method Editor	Why text appears in the bottom box that normally gives information about left-hand side conditions.
Meta-Slot Editor	Why text appears in the top box that normally gives information about the hypothesis.

You can also use the `@V(object.prop)` and `@F(filename)` constructions in the why attribute of all three editors. If a file is specified, it can contain `@V` variables that the system interprets.

Default

If no why text is explicitly specified, the system follows syntactic rules to derive the text displayed by the explanation dialog window.

Inheritance

The Why attribute cannot be inherited.

Related Topics

Rules	Meta-Slots
Methods	Forward Chaining
	Backward Chaining
	Inference

Write Operator

Definition

The *write operator* is used in conditions or actions of rules and methods to write information to a database.

Operands

The *write operator* takes two operands:

- The first operand is either a string constant or an interpretation evaluating to a string constant specifying the name of the file containing the database to be updated or the login name/password for a DBMS.
- The second operand consists of a series of parameters defining the specific update operation to be performed.

Parameters

The second operand may include the following parameters:

@TYPE	Type of database (creator software and file format)
@BEGIN	Command string for opening transaction
@END	Command string for closing transaction
@QUERY	Command string for updating database
@ARGS	Argument list for update command
@ATOMS	List of objects or properties affected
@NAME	Correspondence between objects and records
@FIELDS	List of field names to update
@PROPS	List of properties to update from
@SLOTS	List of slots to update from
@FILL	Create new records or files
@UNKNOWN	Write UNKNOWN values
@CURSOR	Current position for sequential update

See the Database Integration Guide for further details on the meaning and use of these parameters.

When entering a *write* action in the Rule editor or Method editor, clicking in the space for the second operand displays the Database editor dialog box for specifying the update parameters interactively, rather than by explicitly typing them in as listed above.

Note that private slots passed in the argument @Slots are ignored unless the *Write operator* appears in a method specifically triggered for the slot. See the description of Slots for more information about using private slots.

Effect

The designated information is written to the specified database from the Rules Element knowledge base.

Examples

See the Database Integration Guide for examples of the use of the `Write` operator.

Related Topics

Rules	PropertiesClasses
Methods	Slots
Actions	String Constants
Objects	Retrieve Operator

Also see the Database Integration Guide for more information on database operations.

YEAR Function

Definition

The *YEAR function* is used in expressions to extract the year field of a date or time. The expression can appear on the left-hand side or right-hand side of rules and methods.

Syntax

The function consists of the word `YEAR` followed by a single argument in parentheses:

```
YEAR(d)
```

Argument

The argument may be any expression yielding a date or time result. The expression may include patterns or interpretations.

Result

The function returns an integer result equal to the year field of the argument.

If the argument expression does not produce a date or time value, an error message is posted and the function result is `NOTKNOWN`.

Examples

The following examples illustrate the results of the `YEAR` function:

```
YEAR( DATE( 1492, 10, 12 ) )      = 1492
YEAR( DATE( 1981, 6, 8, 21, 8, 46 ) ) = 1981
YEAR( TIME( 8, 4, 23 ) )         = 0
YEAR( TIME( 3, 6, 11, 22, 34, 17 ) ) = 3
YEAR( "October 12, 1492" )      = NOTKNOWN
```

Related Topics

Expressions	HOUR Function
DATE Function	MINUTE Function
TIME Function	SECOND Function
Patterns	WEEKDAY Function

Interpretations
 MONTH Function
 DAY Function

YEARDAY Function
 NOW Function

YEARDAY Function

Definition

The *YEARDAY function* is used in expressions to find the ordinal day of the year corresponding to a given date. The expression can appear on the left-hand side or right-hand side of rules and methods.

Syntax

The function consists of the word `YEARDAY` followed by a single argument in parentheses:

```
YEARDAY ( d )
```

Argument

The argument may be any expression yielding a date result. The expression may include patterns or interpretations.

Result

The function returns an integer result equal to the ordinal day of the year corresponding to the given date argument. The result ranges from 1 to 366.

If the argument expression does not produce a date value, an error message is posted and the function result is `NOTKNOWN`.

Examples

The following examples illustrate the results of the `YEARDAY` function:

```
YEARDAY ( DATE ( 1492 , 10 , 12 ) )      = 286
YEARDAY ( DATE ( 1981 , 6 , 8 , 21 , 8 , 46 ) ) = 159
YEARDAY ( TIME ( 8 , 4 , 23 ) )         = NOTKNOWN
YEARDAY ( TIME ( 3 , 6 , 11 , 22 , 34 , 17 ) ) = NOTKNOWN
YEARDAY ( "October 12, 1492" )         = NOTKNOWN
```

Related Topics

Expressions
 DATE Function
 TIME Function
 Patterns
 Interpretations
 WEEKDAY Function
 NOW Function

YEAR Function
 MONTH Function
 DAY Function
 HOUR Function
 MINUTE Function
 SECOND Function

Yes Operator

Definition

The *Yes operator* is used in the conditions of a rule or method to test whether a boolean value or boolean expression is `TRUE`.

Operands

The *Yes operator* takes a single operand, which must be either a boolean-valued slot, a list of such slots specified by a pattern, or a boolean expression.

Result

The result produced by the *Yes operator* is simply the value of its boolean operand, `TRUE` or `FALSE` as the case may be. If the operand includes a pattern, the condition tests whether at least one of the values in the corresponding list (for an existential pattern) or all of them (for a universal pattern) are `TRUE`. If the operand is a boolean expression, the result is the same as the value of the resolved expression (either `TRUE` or `FALSE`).

Examples

The following are examples of conditions using the *Yes operator*:

```
Yes    credit_approved
Yes    switch_1.on
Yes    <Switch>.on
Yes    {Switch}.on
```

Related Topics

Rules

Methods

Conditions

Boolean Expressions

Boolean Constants

Patterns

No Operator

Execute Library Routines

This chapter describes the various Execute routines you can use as application design features.

Execute Library Overview

Definition

The Rules Element library of Execute routines has predefined procedures for performing common or useful tasks, built into the system for use with the Execute operator.

Routines

The Rules Element run-time library includes the following routines:

Frame Operations

SetValue	CreateObjects
ResetFrame	GetRelatives
CopyFrame	PropagateValue

Multi-Value Operations

AtomNameValue	TestMultiValue
SetMultiValue	ComputeMultiValue
GetMultiValue	LinkMultiValue

Sorting and Comparison

RankList	PatternMatcher
GetListElem	Unify
FindListElem	

Session Control

ControlSession	Message
Journal	WriteTo

Utility Operations

AtomExist	FileExist
CreateReport	Parse

Each of these routines is fully described in its own section of this manual.

Multi-Values

A multi-value slot is defined as a string slot containing a list of values separated by commas. Leading and trailing blanks around each value are ignored, but internal blanks are not. For example, the following is a legal multi-value string:

```
apple, banana, two words, hello
```

This contains four values: `apple`, `banana`, `two words`, and `hello`. Notice that the blanks before and after each value are ignored, but the internal blank in `two words` is retained. Also notice that when a multi-value appears in an expression, it does not have to be enclosed in quotes.

The values are always maintained as strings, but they can be compared as floats, ints, dates, etc. with the `TestMultiValue` `Execute`. But since the values are actually maintained as strings, it is still up to the application developer to make sure the values make sense. In other words, if the application developer wants to do integer comparisons, it is up to him or her to make sure the values really are integers. As far as the Rules Element is concerned, a multi-value slot is just a string slot. See also the section on Using Multi-Values.

Error Handling

Certain global flags can be used to control the handling of errors and tracing information by the built-in `Execute` routines. Currently, the application developer can define “System Objects” to set the error handling and tracing status. At present, the following system objects are implemented:

<code>SYS_ALERTFLAG</code>	Boolean - if true, errors are reported with alert handler.
<code>SYS_TRACEFLAG</code>	Boolean - if true, report trace messages in transcript.
<code>SYS_TRANSFLAG</code>	Boolean - if true, errors are reported in transcript
<code>SYS_BEEPFLAG</code>	Boolean - if true, errors just beep
<code>SYS_STOPFLAG</code>	Boolean - if true, stops session on error.

All of these are boolean-valued objects whose `Value` properties contain the relevant flag. These objects are defined in a separate knowledge base so that they can be loaded in any session. Be sure to use them when developing the application.

Other Notes

- The executes all evaluate to `TRUE` if successful, and `FALSE` if there were any errors.
- Throughout this chapter, the word “frame” is used for “object or class” to describe the `Execute` routines.
- If a slot is expected in a parameter, and you are using a slot with `.Value`, you must explicitly add the `.Value`. Otherwise, the `Execute` routine will assume you are referring to a frame.
- When typing text parameters into the `execute` dialogs, quotes are never used.

Note: The total length of a multi-value is limited only by the available memory.

Invocation

`Execute` routines are invoked by using the `Execute` operator in a condition or action of a rule or method. The first operand to this operator is a string constant giving the name of the desired `Execute` routine; the second operand is a string consisting of a series of parameters to control the routine’s operation.

Parameters

Two standard parameters are used to specify the arguments of an Execute routine (both parameters may be given as dynamic interpretations):

- The `@STRING` parameter passes a single string argument. If two or more such arguments are needed, they can be combined to form a multivalue and passed as a single argument; see the section “Multivalues” for more information.

Atom names you specify for the `@STRING` parameter must be compiled in the corresponding Rules Element editor before the system will recognize it. Merely typing atom names into the execute dialogs’ `@STRING` fields will produce error messages during application processing.

- The `@ATOMID` parameter passes a list of objects, properties, or classes (typically specified via a pattern) for the Execute routine to operate on.

Note: Private slots must not be passed in the `@ATOMID` and `@STRING` parameter of the Execute routines. Also, class name atoms you specify in the execute dialogs must not include vertical bars.

The specific usage of these parameters varies from one Execute routine to another, and is described in the sections on each individual routine.

Result

All Execute routines return a result of `TRUE` if the call is successful, `FALSE` if an error occurs.

Dynamic Values

Individual atoms (objects and object properties) can be evaluated dynamically within the `@STRING` and `@ATOMID` parameters. Each parameter uses its own syntax as follows: `@STRING` interpretations must be in the form of `@V(theAtom.property)` - the atom name enclosed within parentheses and preceded by the characters `@V`. `@ATOMID` interpretations **must** be in the form of `\theAtom.property\` - the atom name enclosed within backslashes. The slot’s current value will then be substituted into the corresponding parameter before execution.

For example, if `Ducks.start` contains the multi-value string `Donald,Daisy` and `Ducks.more` contains `Huey,Dewey, Louie`, then a condition or action of the form

```
Execute "ComputeMultiValue" @ATOMID=Ducks.start;
                                @STRING="@VALUE=@V(Ducks.more),
                                @UNION,@RETURN=Ducks.all";
```

is equivalent to

```
Execute "ComputeMultiValue" @ATOMID=Ducks.start;
                                @STRING="@VALUE=Huey,DeweyLouie,
                                @UNION,@RETURN=Ducks.all";
```

and will set the value of `Ducks.all` to the string `Donald, Daisy,Huey,Dewey,Louie` (the union of `@Ducks.start` and `@Ducks.more`).

When an Execute routine is invoked from a method, atoms can also be evaluated dynamically within the @STRING parameter using the @SELF operator. For example, suppose there is a class `Birds` with a subclass `Ducks`. In addition, suppose `Birds` has a property `Parents` which is a multivalue string and it has an `Order of Sources` method with the following Execute routine:

```
Execute "GetRelatives" @ATOMID=SELF; @STRING="@PARENTS,
@RETURN=@SELF.Parents"
```

If a rule dynamically creates an object called `Donald` of class `Ducks`, and then tries to get the value of `Donald.Parents`, the `Order of Sources` method inherited from `Birds` will be triggered, and `GetRelatives` will evaluate `SELF` as `Donald`. So, `Donald.Parents` will get the multivalue `Ducks, Birds` since these are the parents of `Donald`.

Strategy Options

Many Execute routines include an optional parameter named @STRAT as part of their @STRING parameter. This parameter is used to control the volunteering strategy for any value assignments made during the routine's execution. It can be set to any of the following options:

SET	Store value immediately, but do not forward
FWRD	Queue value for later forwarding if global strategy Forward action effects is currently enabled
SETFWRD	Combines both SET and FWRD options

If no explicit @STRAT parameter is specified, the SET option is assumed by default.

Note: See Chapter One, "Application Development Features" for details on the Strategy operator.

Related Topics

Conditions	Execute Operator
Actions	Patterns
Rules	Value Property
Methods	Multi-Values
String Constants	Inference Strategy

Also see the sections on individual Execute routines by name, as listed above.

Using The Execute Library

The functions in the execute library can be used like any user-defined Execute routine in either conditions or actions of rules and methods. They normally return `TRUE` unless there was some sort of error. They can be divided up into several functional groups:

Frame Operations	This set of routines performs "crunching" operations on frames such as setting values, copying values, etc.
------------------	---

Multi-Value Operations

This set of routines performs operations on multi-values.

Sorting and Comparison

This set of routines performs operations on pattern matching lists.

Session Control

This set of routines controls the session and perform I/O.

Utility Operations

This set of routines performs useful tasks that extend application development.

The following sections explain each of the categories of executes with examples on how you might want to use them.

Frame Operations

The Frame Operations perform “crunching” operations on frames (objects or classes) such as setting values, copying values, etc. They include the following:

`CopyFrame`, `CreateObjects`, `ResetFrame`, `SetValue`, `GetRelatives`, `PropagateValue`

These operations do things which could be done frame by frame in other ways, but it is more convenient to use these executes. For example, `CopyFrame` copies the values in all properties of a frame (except `Value`) to a list of frames. Without this function, you could copy the values one by one, but it would be very inconvenient.

Also, the `Reset` operator could be used to reset individual slots, but the `ResetFrame` execute can reset all the slots in a list of frames all at once.

`CreateObjects` eliminates the need to have a rule which loops around itself creating objects one by one.

`SetValue` sets all slots in a list of slots or frames to a given value, which again, would be very inconvenient otherwise.

`GetRelatives` gets the names of the parents or children of a frame and returns the answer as a multi-value.

`PropagateValue` propagates a value up or down through the inheritance paths from a given frame.

Multi-Value Operations

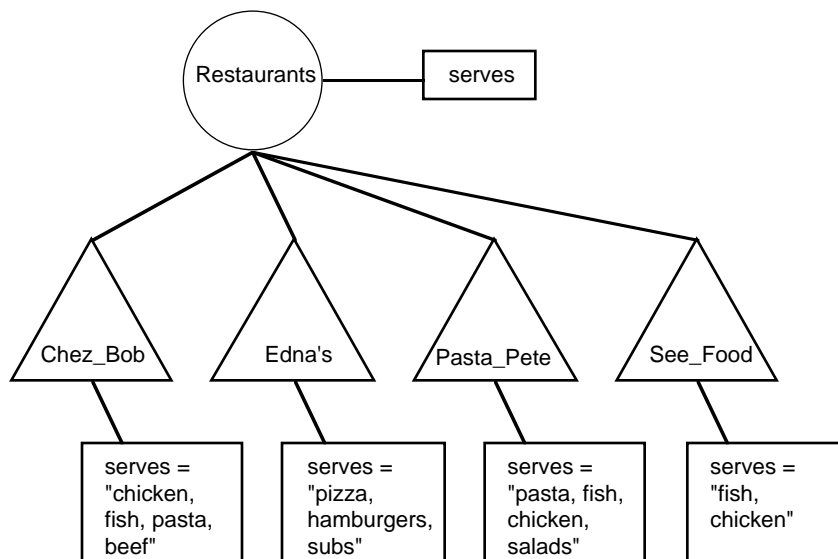
Multi-values can be used in many ways. The executes that deal with multi-values are as follows:

`ComputeMultiValue`, `GetMultiValue`, `LinkMultiValue`, `SetMultiValue`, `TestMultiValue`, `AtomNameValue`

One way you might want to use multi-values is to keep track of properties which have an unspecified number of “sub-properties”. For example, you might have a class of `<Restaurants>` with a property `serves` which contains the types of food served at a certain restaurant. The `serves` property for a given restaurant might contain something like `chicken`, `fish`, `pasta`. The `SetMultiValue` execute can be used for adding and deleting values from these multi-values. This is an ideal way of maintaining

this information because each restaurant may serve a different number of foods.

The following diagram shows an example of how this sort of example might be set up:



Now, suppose you wanted to ask something like, “Who serves fish?” You could use the `TestMultiValue` execute to find all the restaurants that serve fish and attach them to a class like this:

```
TestMultiValue (@STRING="@TEST=fish, @SUPERSET,
@RETURN=Fishy";
               @ATOMID=<Restaurants>.serves;)
```

After that execute, the restaurants `Chez_Bob`, `Pasta_Pete`, and `See_Food` will be attached to the class `Fishy`. You could then do further pattern matching or testing on that list. Notice we are using `@SUPERSET` because we are finding the restaurants that serve a superset of fish. A restaurant which serves *only* fish would qualify.

Now let’s ask the question, “What do `Chez_Bob` and `Pasta_Pete` have in common?” We would do that like this:

```
ComputeMultiValue (@STRING="@VALUE=@V(Chez_Bob.serves),
@INTERSECT, @RETURN=common.mulval";
                  @ATOMID=Pasta_Pete.serves;)
```

After that execute, the multi-value `common.mulval` will contain the intersection of the two restaurants, i.e. `chicken, fish, pasta`. Notice we are using the `@V(...)` notation to evaluate `Chez_Bob.serves` dynamically.

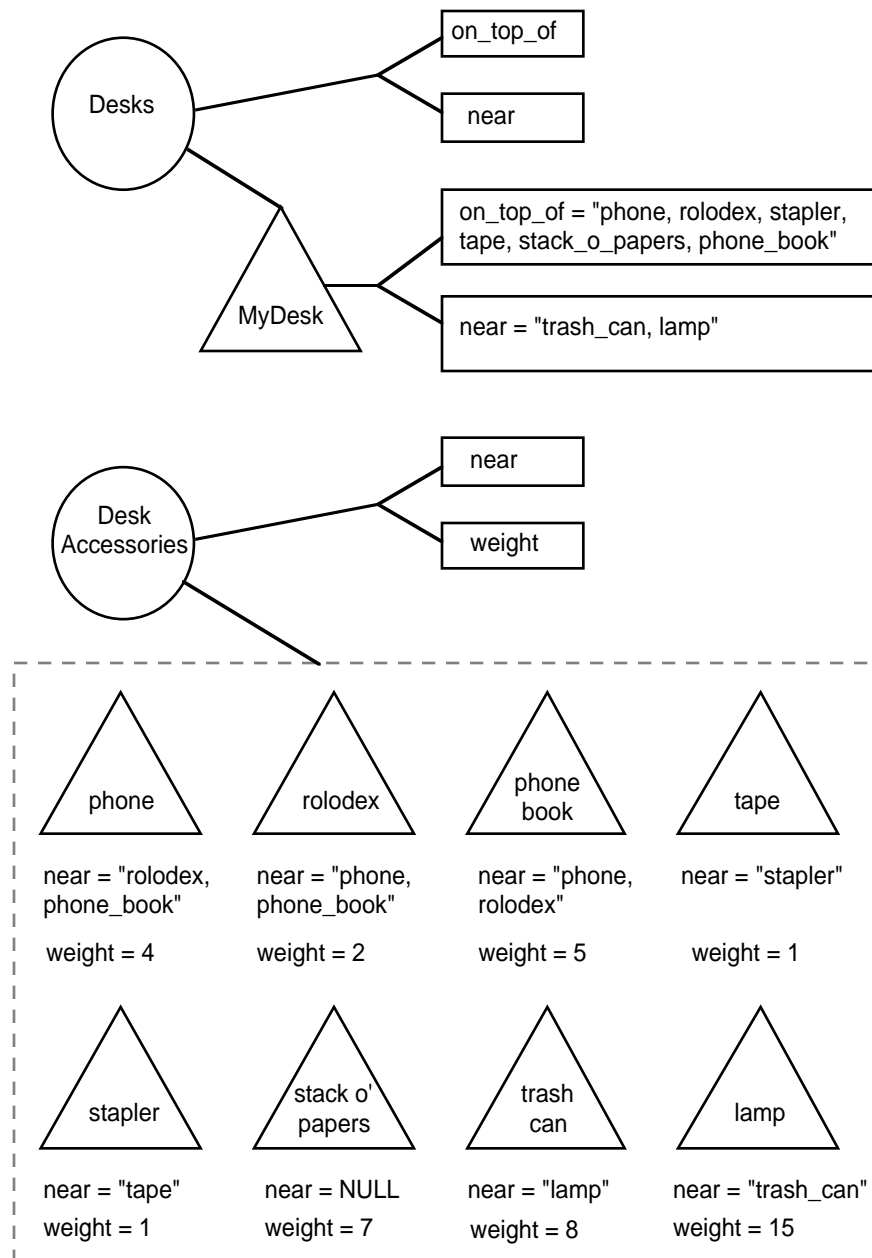
Now let’s say that `Pasta_Pete` is purchased by `Chez_Bob`, so they decide to combine the menus. We could add the foods served by `Pasta_Pete` to the foods served by `Chez_Bob` like this:

```
SetMultiValue (@STRING="@ADD=@V(Pasta_Pete.serves)";
               @ATOMID=Chez_Bob.serves;)
```

Since `@NODUPLICATE` is the default, `Chez_Bob.serves` will now contain `chicken, fish, pasta, beef, salads`. Again, notice the `@V(...)` notation.

Another way you might want to use multi-values is to maintain relationships between objects. The values within a multi-value could actually be object names. So, for example, you might have a class of <Desks> with a property `on_top_of`, and another class of <Desk_accessories>. The `on_top_of` slot for a given desk might contain the names of <Desk_accessories> objects which are on top of the desk. So, a given desk may have an `on_top_of` slot containing something like `stapler`, `tape`, `phone`. Each of the values in that multi-value are actually the names of objects in the class <Desk_accessories>.

The following figure shows a possible configuration with multi-values being used for relationships between objects:



With this sort of set up, there are all sorts of questions we might want to ask. For example, “What’s on top of my desk?” This may seem trivial, but it could be that the multi-value was constructed by other rules, and you may now want to use that list as a pattern matching list in another condition of a rule or method. So, to get the objects in a multi-value and attach them to a class, we would do this:

```
LinkMultiValue (@STRING="@LINKTO=DeskStuff";@ATOMID=MyDesk.on_top_of;)
```

After this execute, the class `DeskStuff` would have as objects `phone`, `rolodex`, `stapler`, `tape`, `stack_o_papers`, and `phone_book`.

Another thing you might want to do is construct a multi-value containing the names of all the desk accessories. That multi-value could then be used with `TestMultiValue` or `ComputeMultiValue`. This would be done like this:

```
AtomNameValue
(@STRING="@RETURN=DeskStuff.mulval";@ATOMID=<Desk_Accessories>;)
```

After this execute, `DeskStuff.mulval` will contain the multi-value `phone`, `rolodex`, `phone_book`, `tape`, `stapler`, `stack_o_papers`, `trash_can`, `lamp`.

You might also want to ask more complicated questions like, “What objects on my desk are heavy?” Let’s assume that “heavy” is greater than or equal to five pounds. To do this, we would first need to create a list of objects on the desk by using `LinkMultiValue` as above. Then, we would use the list `<DeskStuff>` in a pattern matching statement like this:

```
>= <DeskStuff>.weight 5
```

Directly after this statement, the pattern matching list will contain only the objects on the desk whose weight is greater than or equal to 5.

Another possible question would be, “Is the trash can on top of my desk?” To do this, you would use the following execute:

```
TestMultiValue (@STRING="@TEST=trash_can, @SUPERSET,
                 @RETURN=answer.bool"; @ATOMID=MyDesk.on_top_of;)
```

After this execute, the boolean slot `answer.bool` will contain `FALSE` because the multi-value `MyDesk.on_top_of` is not a superset of `trash_can`. Or, to put it another way, `MyDesk.on_top_of` does not contain `trash_can`.

Now, suppose there is an earthquake and the phone falls off the desk. How would we update our objects to reflect this? First, we want to remove the phone from the desk, and then we want to update the objects that the phone is near, and the objects that are near the phone. There is probably more than one way to do this, but here is one possibility:

Step 1: Remove phone from the desk:

```
SetMultiValue (@STRING="@DELETE=phone";
@ATOMID=MyDesk.on_top_of;)
```

Step 2: Link the objects that were near the phone to a temporary class:

```
LinkMultiValue (@STRING="@LINKTO=NearStuff";
@ATOMID=phone.near;)
```

Step 3: Make sure none of those objects is near the phone:

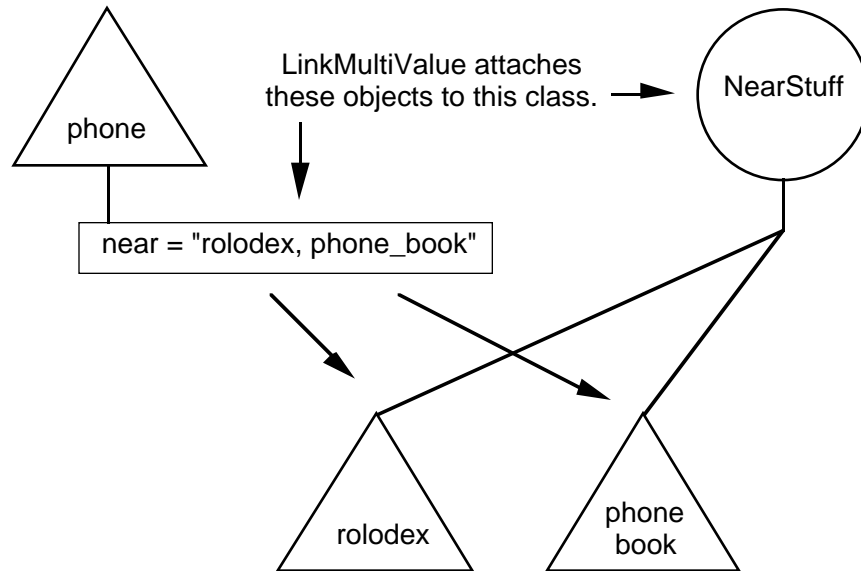
```
SetMultiValue (@STRING="@DELETE=phone";
@ATOMID=<NearStuff>.near;)
```


Step 4: Delete all the things that were near the phone:

```
SetMultiValue (@STRING="@DELETE=@V(phone.near)"; @ATOMID=phone.near;)
```

Okay, so how does this work? Step 1 simply deletes phone from the multi-value `MyDesk.on_top_of`.

Step 2 takes the object names in the multi-value `phone.near` and links them to a temporary class `NearStuff`. In this case, that would link the objects `rolodex` and `phone_book` to `NearStuff` like this:



Step 3 deletes `phone` from each of the multi-values in the list `<NearStuff>.near`. In other words, since the `phone` is not near any of the objects in `<NearStuff>` anymore, we want to make sure that those objects do not list `phone` as a nearby thing. So, in this case, `phone` is deleted from `rolodex.near` and `phone_book.near`.

Finally, step 4 deletes everything that was near the phone because it is not near anything anymore. Notice we are using the `@V(...)` notation to insure that everything in the current multi-value is deleted from itself. For this step, you could also simply set the value of `phone.near` to an empty string using the `Assign` operator. Notice that this is not the same as setting it to `UNKNOWN`. Notice also that if you use the `Assign` operator, you may cause side effects like forwarding through gates unless you set the strategies appropriately.

Sort and Compare

The `Sorting` and `Comparison` executes perform operations on pattern matching lists. This category includes the following executes:

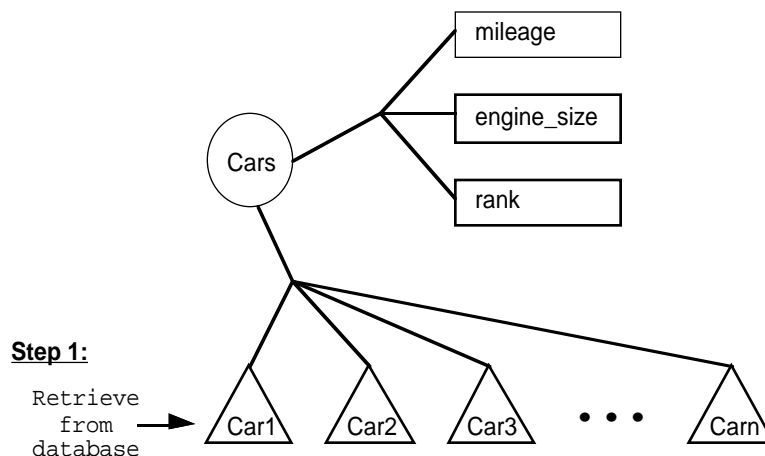
```
FindListElem, GetListElem, RankList, PatternMatcher, Unify
```

These are used for ranking lists and getting individual elements, ranges of elements, or finding the `MIN` and `MAX` in a list. Also, `PatternMatcher` performs a more general purpose pattern matching, and `Unify` performs a two-way pattern match.

For example, suppose we have a class of `<Cars>` with properties `mileage`, `engine_size` and `rank`. In a database, we have the latest information on

current cars. So, we create an object for each car and get the mileage and engine_size from the database. Now, suppose we want to find the ten highest mileage cars available. To do this, we would first use the RankList execute to rank the list using mileage as the RANKBY property, and rank as the RANKSET property. Then, we would use GetListElem to get elements 1 through 10 by rank. We could then use the returned list of ten cars to do some other pattern matching like finding the cars whose engines are greater than or equal to 1500 cc.

The following diagram shows the steps involved in this example:



Step 1:

Retrieve
from database →

Step 2:

```
Execute "RankList" (@STRING="@RANKBY=mileage, @RANKSET=rank,
@DECREASING"; @ATOMID=<Cars>;)
```

Step 3:

```
Execute "GetListElem" (@STRING="@FROM=1, @TO=10, @RANKSET=rank,
@LINKTO=High_Mileages"; @ATOMID=<Cars>;)
```

Step 4:

```
>= <|High_Mileages|>.engine_size 1500
```

Session Control

The Session Control operations control the session and perform I/O. They include the following:

ControlSession, Message, Journal, WriteTo

A very useful thing that you can do is send messages to the environment. For example, you can put results into the transcript, or issue alert messages to the user. You can also put up question boxes in which the user must respond with Yes, No, OK, or Cancel. You can then use the response to control the application. If you are writing your own environment in C or some other language, these executes will call your own transcript handler or alert handler. The Session Control operations also control the session, suggest hypotheses, perform journaling, and so on.

Utilities

The Utilities are Execute routines that perform useful tasks that extend your application development capabilities. They include the following:

AtomExist, CreateReport, FileExist, Parse

Generally, these functions are used for testing the existence of certain things, and sending messages to the environment. For example, suppose you have a Retrieve in your rule, but the file does not exist. Normally, you would get an error, and the rule would simply fail at that point. But, by using FileExist before the Retrieve, you could check if the file exists and then act accordingly. For example, if the file doesn't exist, you might want to try another file, or a different search path. Another particularly useful routine is CreateReport. This routine lets you generate a formatted file to report the results of an application processing session.

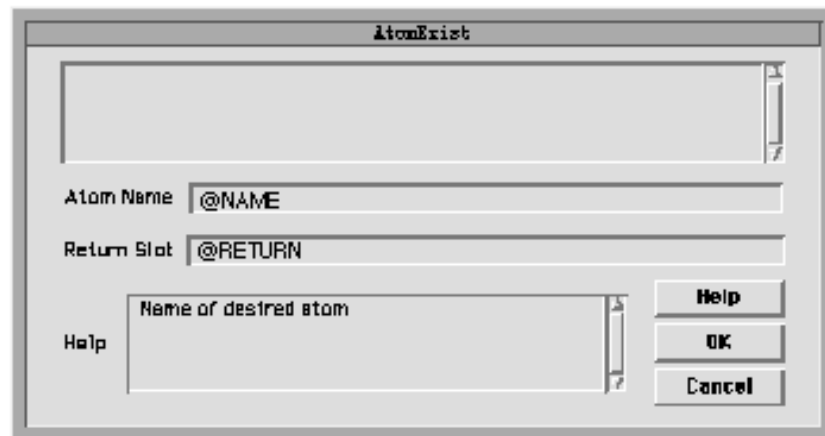
AtomExist Routine

Definition

The Execute routine AtomExist tests whether a designated atom (a class, object, property, slot, rule, or method) currently exists.

Interactive Dialog

AtomExist is chosen with the Select Execute popup menu command in the Rule editor or Method editor, which automatically displays a special dialog box for specifying the execute parameters interactively, rather than by explicitly typing them in as listed below:



Parameters

The @ATOMID parameter is ignored.

The @STRING parameter must include the following:

@NAME=*atom_name* Name of desired atom.

@RETURN=*answer_slot* Name of slot in which to return result of test.

Both parameters are required. The destination specified by @RETURN must be a boolean-valued slot.

Effect

The destination slot designated by the @RETURN parameter is set to TRUE or FALSE, depending on whether the requested atom currently exists.

Result

The result returned by AtomExist is TRUE if the call is successful, FALSE if an error occurs.

Examples

A condition or action of the form

```
Execute "AtomExist"           @STRING="@NAME=Flapdoodle,
                              @RETURN=TheAnswer.Value";
```

will set TheAnswer.Value to TRUE if the object Flapdoodle currently exists, FALSE if it does not.

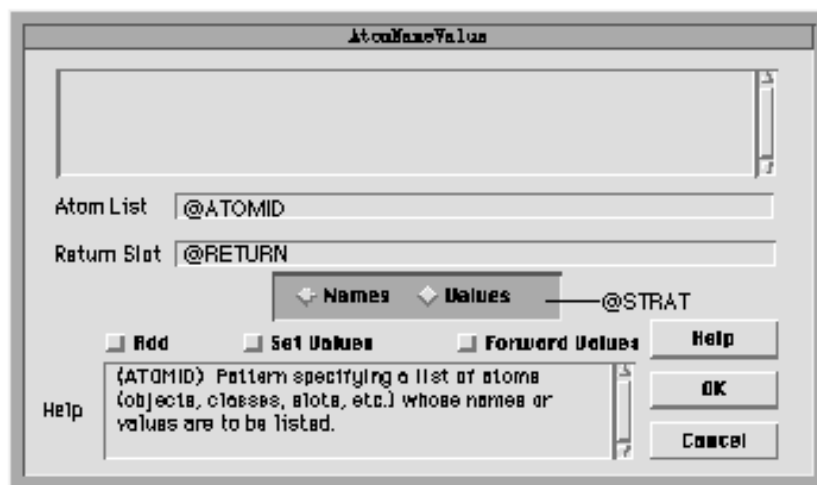
AtomNameValue Routine

Definition

The Execute routine AtomNameValue stores the names or values of one or more atoms (objects, classes, or slots) into a string-valued variable as a multivalue.

Interactive Dialog

AtomNameValue is chosen with the Select Execute popup menu command in the Rule editor or Method editor, which automatically displays a special dialog box for specifying the execute parameters interactively, rather than by explicitly typing them in as listed below:



Parameters

The @ATOMID parameter is a pattern specifying a list of atoms (objects, classes, or slots) whose names or values are to be listed.

The @STRING parameter may include the following:

@RETURN= <i>destination</i>	String slot into which the requested atom names or values are to be stored.
@ADD	(Optional) If present, append new atom names or values to existing contents of destination variable instead of assigning outright (no duplicates are added).
@STRAT= <i>options</i>	(Optional) Strategy options governing the assignment (see Execute Library Overview for details).
@NAMES	(Optional) The names of the atoms are listed in the destination. This is the default.
@VALUES	(Optional) The values of the atoms are listed in the destination. In this case, the atoms must be slots.

The @RETURN parameter must designate a slot (property associated with an object) as the destination, and not simply the name of an object itself: for example, a destination of @RETURN=theResult is invalid, and must be specified as @RETURN=theResult.Value instead.

Effect

The names or values of the atoms satisfying the pattern given by @ATOMID are concatenated together, separated by commas, to form a multi-value. (Notice that if there is only one such atom, its name alone is equivalent to a one-element multi-value.) This multi-value is then assigned as the new value of the string variable designated by @RETURN (unless @ADD is specified, in which case it is instead appended to the end of the variable's existing value without duplicates. Duplicates can be requested explicitly with the SetMultiValue routine).

Result

The result returned by AtomNameValue is TRUE if the call is successful, FALSE if an error occurs.

Examples

If class Duck has a single instance named Donald, a condition or action of the form

```
Execute "AtomNameValue" @ATOMID=<Duck>;
@STRING="@RETURN=Duckburg.residents";
```

will assign the string Donald as the value of Duckburg.residents. If there are two instances of Duck named Donald and Daisy, Duckburg.residents will be set to the multi-value string Donald,Daisy. If the previous value of Duckburg.residents was Daffy, then

```
Execute "AtomNameValue" @ATOMID=<Duck>;
@STRING="@RETURN=
Duckburg.residents,
@ADD";
```

will set it to Daffy, Donald, Daisy. If the object Nephews has three components (subobjects) named Huey, Dewey, and Louie, then

```
Execute "AtomNameValue"    @ATOMID=<Nephews>.uncle;
                          @STRING=
"@RETURN=Cartoon relatives";
```

will set Huey.uncle, Dewey.uncle, Louie.uncle as the new value of Cartoon.relatives.

If class Duck has two instances, Donald and Daisy, and a property bill_size, and Donald.bill_size is 5 and Daisy.bill_size is 4, then:

```
Execute "AtomNameValue"    @ATOMID=<Duck>.bill_size;
                          @STRING="@RETURN=Duckbill.sizes,
                          @Values";
```

will set the string slot Duckbill.sizes to the multivalue string 5, 4. The value type of the slots in @ATOMID can be anything (string, integer, time, date, etc.)

Related Topics

Execute Operator

Multi-Values

Patterns

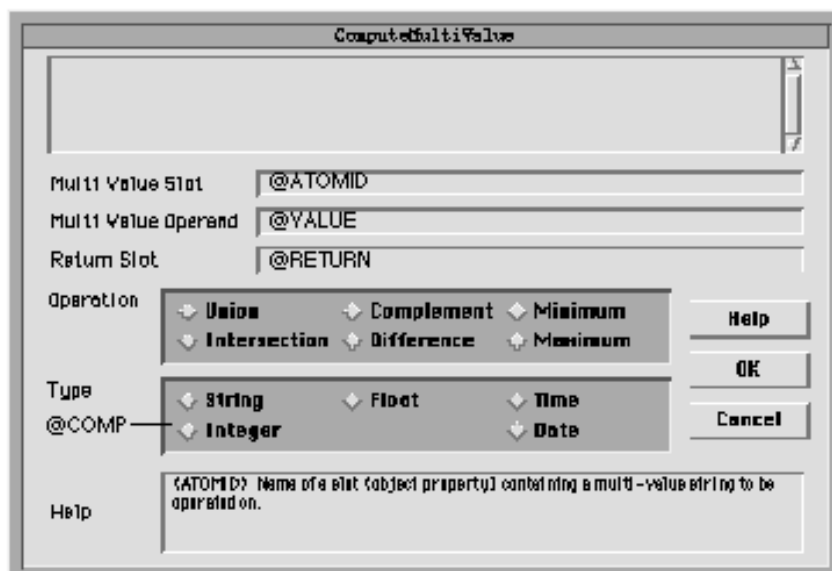
ComputeMultiValue Routine

Definition

The Execute routine ComputeMultiValue combines multi-values in various ways to form new multi-values.

Interactive Dialog

ComputeMultiValue is chosen with the Select Execute popup menu command in the Rule editor or Method editor, which automatically displays a special dialog box for specifying the execute parameters interactively, rather than by explicitly typing them in as listed below:



Parameters

The @ATOMID parameter is the name of a slot (a property associated with a given object) containing a multi-value string to be operated on.

The @STRING parameter may include the following:

@VALUE= <i>multi_val</i>	(Required for some operations--see Operations below) Second multi-value operand.
@ <i>operation</i>	Operation to be performed (see Operations below).
@RETURN= <i>answer</i>	Destination slot in which to return result of operation.
@COMP= <i>value-type</i>	(Optional) Specifies the way in which the individual values in the multivalues are to be compared (see Value Types below).
@VALUETYPE= <i>type</i>	The valuetype specifier can be used for indicating how the individual values in a multivalued are to be compared. If it is absent, STRING is the default.

Operations

The *operation specifier* included in the @STRING parameter identifies the operation to be performed on the pair of multi-values designated by @ATOMID and @VALUE. It must consist of exactly one of the following:

@UNION	All elements in either @ATOMID or @VALUE or both
@INTERSECT	All elements in both @ATOMID and @VALUE
@COMPLEMENT	All elements in @ATOMID or @VALUE but not both
@DIFFERENCE	All elements in @ATOMID but not @VALUE
@MIN	Smallest element in @ATOMID
@MAX	Largest element in @ATOMID

Notice that the operations @MIN and @MAX take only one operand (@ATOMID); the second operand (@VALUE) is ignored and may be omitted.

Value Types

The @COMP specifier can be used for indicating how the individual values in a multivalued are to be compared. If it is absent, STRING is the default. The following types are valid: STRING, INT, FLOAT, DATE, and TIME.

For example, if one multivalued contains the element 1.0 and another multivalued contains the element 1.00, these will be regarded as the same value if @COMP=FLOAT is specified. However, if @COMP=STRING is specified (the default), they are regarded as two different strings.

Effect

The two multi-values specified by the @ATOMID and @VALUE parameters are combined according to the requested operation, and the result is stored into the destination slot designated by @RETURN.

Result

The result returned by `ComputeMultiValue` is `TRUE` if the call is successful, `FALSE` if an error occurs.

Examples

If `Ducks.start` contains the multi-value `Donald, Daisy, Dewey`, a condition or action of the form

```
Execute "ComputeMultiValue"@ATOMID=Ducks.start;
      @STRING="@VALUE=Huey,Dewey,Louie,
      @UNION,@RETURN=Ducks.union";
```

will assign the string `Donald, Daisy, Dewey, Huey, Louie` (the union of `@ATOMID` and `@VALUE`) as the new value of `Ducks.union`; notice that the element `Dewey` is not duplicated.

```
Execute "ComputeMultiValue"@ATOMID=Ducks.start;
      @STRING="@VALUE=Huey,Dewey,Louie,
```

```
@INTERSECT,@RETURN=Ducks.intersect";
```

will set `Ducks.intersect` to `Dewey` (the intersection of `@ATOMID` and `@VALUE`).

```
Execute "ComputeMultiValue"@ATOMID=Ducks.start;
```

```
@STRING="@MIN,@RETURN=Ducks.first";
```

will set `Ducks.first` to `Daisy` (the smallest element alphabetically in `@ATOMID`).

Related Topics

Execute Operator
Multi-Values

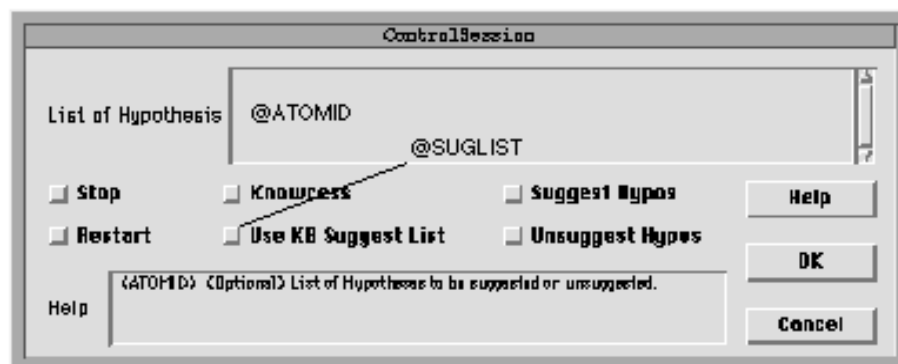
ControlSession Routine

Definition

The `Execute` routine `ControlSession` performs various control operations affecting the operation of the current Rules Element session.

Interactive Dialog

`ControlSession` is chosen with the `Select Execute` popup menu command in the Rule editor or Method editor, which automatically displays a special dialog box for specifying the execute parameters interactively, rather than by explicitly typing them in as listed below:



Parameters

The @ATOMID parameter is an (optional) list of hypotheses to be suggested or unsuggested.

The @STRING parameter may include the following:

@STOP	(Optional) Stop session.
@RESTART	(Optional) Stop session and reinitialize all values.
@SUGLIST	(Optional) Suggest hypotheses on knowledge base's suggest list.
@SUGGEST	(Optional) Suggest hypotheses specified by @ATOMID.
@UNSUGGEST	(Optional) Unsuggest hypotheses specified by @ATOMID.
@KNOWCESS	(Optional) Initiate inference processing.

The parameters @SUGGEST and @UNSUGGEST are mutually exclusive, and may not both be specified. If neither is present, the @ATOMID parameter is ignored.

Effect

The control operations specified by the @STRING parameter are executed. Operations are always performed in the order shown under "Parameters" above, regardless of the order in which they actually appear in the @STRING parameter.

All parameters in ControlSession are performed even if one of them is StopSession. ControlSession can be regarded as a single atomic function.

The operations @SUGGEST and @UNSUGGEST apply to the list of hypotheses specified by the @ATOMID parameter; @SUGLIST applies to the hypotheses in the suggest list saved with the knowledge base itself.

The operations @RESTART and @KNOWCESS are equivalent to the Expert menu commands Restart Session and Knowcess, respectively.

Result

The result returned by ControlSession is TRUE if the call is successful, FALSE if an error occurs.

Examples

A condition or action of the form

```
Execute "ControlSession" @STRING="@STOP";
```

will stop the current session.

```
Execute "ControlSession" @STRING="@RESTART, @SUGLIST, @KNOWCESS";
```

will stop the session, reinitialize all values, suggest all hypotheses on the knowledge base's suggest list, and restart inference processing.

```
Execute "ControlSession" @STRING="@SUGGEST";
@ATOMID=hypo1,hypo2;
```

will suggest the hypotheses hypo1 and hypo2.

Related Topics

Multi-Values

Patterns

Execute Operator

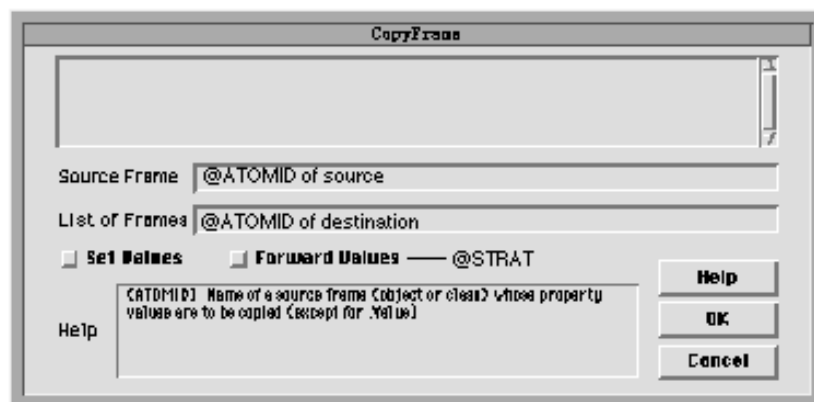
CopyFrame Routine

Definition

The Execute routine `CopyFrame` copies property values from one frame (object or class) to another.

Interactive Dialog

`CopyFrame` is chosen with the Select Execute popup menu command in the Rule editor or Method editor, which automatically displays a special dialog box for specifying the execute parameters interactively, rather than by explicitly typing them in as listed below:



Parameters

The `@ATOMID` parameter consists of two items:

- The name of a source frame (object or class) whose property values are to be copied
- The name of the destination frame to which they are to be copied, or a pattern specifying a list of such frames

The @STRING parameter is optional, and consists entirely of the following:

@STRAT=*options* (Optional) Strategy options governing the assignment (see Execute Library Overview for details).

Effect

The values of all of the source frame's properties are copied to the corresponding properties of each destination frame, with the following exceptions:

- The destination frame must already possess a property with the given name. If it does not, the property is not automatically associated with the destination frame and its value is not copied.
- The source frame's Value property is never copied.

Result

The result returned by CopyFrame is TRUE if the call is successful, FALSE if an error occurs.

Examples

Suppose class Box has two instances named box1 and box2 and two properties named width and height, and that cube1 is an instance of class Cube with properties width, height, and depth. Then a condition or action of the form

```
Execute "CopyFrame" @ATOMID=cube1,box1;
```

will copy the values of cube1.width and cube1.height to box1.width and box1.height, respectively. The value of cube1.depth is not copied, since the destination frame box1 has no property named depth.

```
Execute "CopyFrame" @ATOMID=cube1,<Box>;
```

will set both box1.width and box2.width equal to cube1.width, and both box1.height and box2.height equal to cube1.height.

Related Topics

Execute Operator
Data Types
Value Property
Patterns

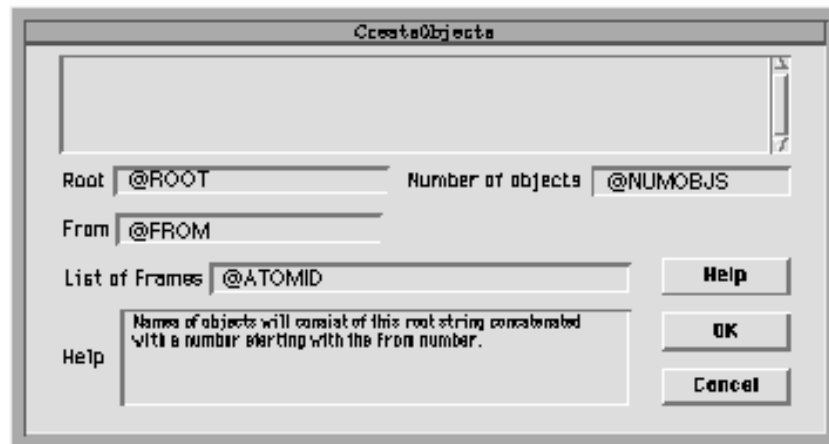
CreateObjects Routine

Definition

The Execute routine CreateObjects creates dynamic objects and attaches them to one or more frames (classes or objects) as specified.

Interactive Dialog

`CreateObjects` is chosen with the Select Execute popup menu command in the Rule editor or Method editor, which automatically displays a special dialog box for specifying the execute parameters interactively, rather than by explicitly typing them in as listed below:



Parameters

The `@ATOMID` parameter is the list of frames (objects or classes) whose properties the dynamic objects may inherit.

The `@STRING` parameter may include the following:

- `@ROOT=obj_name` Root part of name assigned to all created objects, full name includes `start_num`.
- `@FROM=start_num` (Optional) System increments this starting number for each new dynamic object and adds to root part of name to create unique object name.
- `@NUMOBS=total` Number of dynamic objects the system is to create.

The full object name consists of the concatenated values of `@ROOT` and `@FROM`. If `@FROM` is omitted, the system automatically increments the number part of the object name starting from the default value 1.

Effect

The parameters `@ROOT` and `@FROM` (if present) determine the name of objects the system creates dynamically by attaching them to the parent objects or classes specified in `@ATOMID`. The system keeps track of the total number of objects created by incrementing the number part of the full object name and stops when the number reaches the specified number `@NUMOBS`. Dynamic objects automatically inherit properties from their parents if the inheritance strategy is unmodified.

Result

The result returned by `CreateObjects` is `TRUE` if the call is successful, `FALSE` if an error occurs.

Examples

A condition or action of the form

```
Execute "CreateObjects"    @STRING="@ROOT=myObj, @NUMOBS=10";
                          @ATOMID=ClassA,ClassB;)
```

will create ten objects called myObj1 through myObj10. Each of them will be attached to the classes ClassA and ClassB. If any of these objects already exist, they will just be attached to the classes. If the @FROM parameter is added to the previous example, then

```
Execute "CreateObjects"    @STRING="@ROOT=myObj, @FROM=21,
                          @NUMOBS=10"; @ATOMID=ClassA;)
```

will create ten objects called myObj21 through myObj30 and attach them each to ClassA.

Related Topics

Properties

Execute Operator

Inheritance Strategy

Dynamic Objects

Inheritance

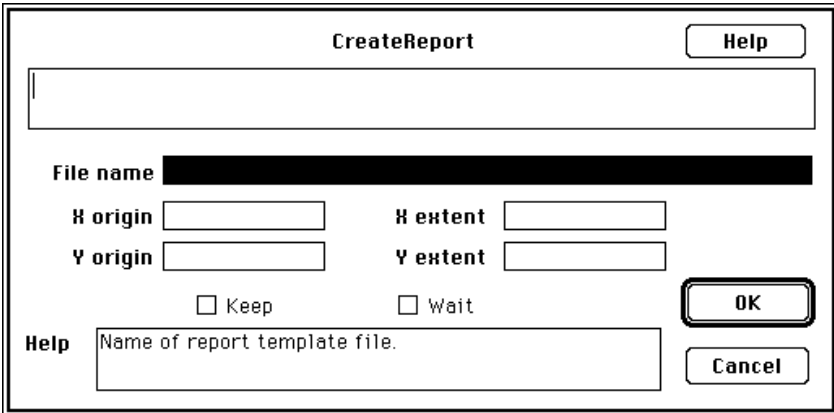
CreateReport Routine

Definition

The Execute routine `CreateReport` processes a text file containing formatting commands and interpretations on slot variables and then displays the processed file.

Interactive Dialog

`CreateReport` is chosen with the Select Execute popup menu command in the Rule editor or Method editor, which automatically displays a special dialog box for specifying the execute parameters interactively, rather than by explicitly typing them in as listed below:



Parameters

The @ATOMID parameter is ignored.

The @STRING parameter can include the following:

@FILE= <i>file_name</i>	Name of file to convert.
@ORX= <i>horizontal origin</i>	Horizontal origin of the window displayed.
@ORY= <i>vertical origin</i>	Vertical origin of the window displayed.
@EXTX= <i>width</i>	Width of the window displayed.
@EXTY= <i>height</i>	Height of the window displayed.
@WAIT	Display Continue and Close buttons; Wait for mouse click before continuing.
@KEEP	Display Information in a new window and keep it until the next show or the user explicitly closes it.

Only the FILE parameter is required.

Effect

The text file designated by the @FILE parameter is processed line by line. It can contain commands and slot variable names as described below. You assemble the commands of the text formatting language in the text file using any text editor.

Formatting commands specify alignment, page length, inclusion of other files, and sections which are included or not depending on variable values from the Rules Element. (See Formatting Commands for more information on the available commands.) Slot variables names are interpreted and are substituted with their current values.

The processed file will not be displayed if it contains a #noDisplay# formatting command and if it is saved on disk (#NewFile# or #AddFile# commands).

Interpreting Slot Variables

When CreateReport processes a text file, the contents of the text file are preprocessed. It considers any string between “\” (back slashes) or between the parentheses of @V() an interpretation (dynamic value) and replaces it with the current value of the interpreted variable, provided it is a valid slot of the knowledge base. For example:

```
... \passenger.fullname \...
```

or

```
... @V(passenger.fullname)
```

is displayed as ...Mark Johnson... if the current value of the property fullname of the object passenger is Mark Johnson.

After this preprocessing, the system searches for global commands and executes the corresponding instructions. Afterwards, the system scans the text one character at a time; it then interprets and executes the local commands.

Important:

- All the formatting commands and their arguments can be interpreted. This means that the system can interpret a dynamic Rules Element variable to obtain the command keyword and its arguments.

- If your final text must contain a backslash (\), write “\\” so that the parser does not mistake it for an interpretation.

Because the dynamic values are interpreted before the rest, you should beware of using dynamic values containing “#”: when the system scans the content of the file, it understands these symbols as the beginning or the end of a command.

Formatting Commands

The text formatting language contains commands that describe the way the text following the command needs to appear on the screen. Each command starts and finishes with a # on its own line. Do not use this symbol in the text itself. The following list identifies these commands that belong to one of three categories: screen layout commands, text commands, and file commands.

Screen Layout Commands

The following commands control how much of the screen the text window uses. A text window can consist of several screen pages and is scrollable by the end user.

Specify a Header#OpenHeader#
 ...(text)
 #CloseHeader#

These two commands specify a header for the text window using text you supply. The text must not exceed five lines. The text between these two commands can contain interpretations of Rules Element variables.

Specify a Footer#OpenFooter#
 ...(text)
 #CloseFooter#

These two commands specify a footer for the text window using text you supply. The text must not exceed five lines. The text in between these two commands can contain interpretations of Rules Element variables.

Set Page Length#PageLength=XXX#

This command sets the length of the page (number of lines between the top two consecutive headers) to XXX. The system adds a page break every XXX lines. The default page length is 50 lines. Page breaks appear as lines of “-”. To change this default character, see the #SetPageBreak=char# below.

Set Page Width#PageWidth=XXX#

This command sets the width of the page (in number of columns) to XXX. Lines wrap around every XXX characters. The default page width is 80 columns.

Set Left Margin#LeftMargin=XXX#

This command sets the left margin to XXX characters. The default left margin is 0.

Set Right Margin#RightMargin=XXX#

This command sets the right margin to XXX characters. The default right margin is 0.

Specify Page Break Character`#SetPageBreak=char#`

This command changes the character used for displaying page breaks to the one you specify.

Insert Page Break`#PageBreak#`

This command forces a page break on the line.

Text Commands

The following commands control aspects of the text itself, including color, alignment, and exact position:

Set Tabs`#Tabs=X#`

This command lets you tab at every X number of spaces or multiple of the number. Be careful using tabs with dynamic text variables since the formatted text position depends on the slot value not the slot name.

Center Text`#Center#`

This command centers the text following it. Text remains centered until the Rules Element finds a `#LeftAlign#` or `#RightAlign#` command.

Left Align Text`#LeftAlign#`

This command makes the text following it left aligned. Text stays left aligned until the Rules Element finds a `#Center#` or `#RightAlign#` command.

Right Align Text`#RightAlign#`

This command makes the text following it right aligned. Text stays right aligned until the Rules Element finds a `#Center#` or `#LeftAlign#` command.

Set Text Column`#LXXX#`

This command begins the text following it on column XXX. This command can appear embedded inside the text.

Align Text Column`#RXXX#`

This command begins the text following it on column XXX and makes it right aligned. This command can appear embedded inside the text.

Set Word Wrap`#WordWrap#`

This command allows word wrap. Text you display does not exceed `#PageWidth=XXX#`. Word wrap is the default condition.

Set Character Wrap`#CharWrap#`

This command allows character wrap. This disables the word wrap condition.

Set Precision`#Precision=X#`

This command lets you change the precision used to display the fractional part of a floating point number. The default is 0, so fractions are ignored.

Set Date`#date=YYYYY#`

This command displays the current system date in the format specified by YYYYY, where Y can be any of the following characters:

d	Uses the current date.
h	Uses the current hour.
m	Uses the month number.
y	Uses the current year.
D	Uses the first three letters of the day.
M	Uses the first three letters of the month.

Blank spaces and '/' are valid separators. As an example, #date=D_m/d/y# is replaced by Wed_01/03/90 (the underscore denotes a blank space).

File Commands

The following commands access files or external devices:

Override Form Feed#NoFormFeed#

This command overrides the default form feed that normally occurs when you print a text file, create a new file (#NewFile#), or append the file to an existing file (#AddFile#).

Include a File#Include a filename [<class>] {[+,-] index}

This command causes the file you specify (filename) to appear in the current text file. For complete details about including files, refer to the Include Command section below.

Copy Text to File#NewFile=filename#

This command creates a new text file (filename) and stores all the text preceding this command in the newly created file. The text is stored exactly as it appears on the screen, without commands.

Store Text Only in File#AddFile=filename#

This command stores all the text preceding this command in the file you specify (filename), without commands. If the file you specify already exists, the text is appended to the end of the file.

Do not display the Text#NoDisplay#

This command will cause the text not to be displayed once it has been processed (the default is to display the text). However, this commands will only be effective if the text was saved with a #NewFile# or #AddFile# command.

Conditional Statements

The following command structure lets you display text or execute commands only if the conditions you specify are met.

```
#if( condition )#
... commands and text
#elseif( condition )#
... commands and text
#else#
... commands and text
#endif#
```

The condition compares one or more variables of the knowledge base to the value you specify as follows:

```
(\ObjectName.Property\==Value)
```

This command structure uses the following operators to make comparisons:

```
!=      Variable is not equal to the value.
==      Variable is equal to the value.
<      Variable is less than value.
>      Variable is greater than value.
<=     Variable is less than or equal to value.
>=     Variable is greater than or equal to value.
```

Additionally, logical operators let you chain variables together or negate the variable, as follows:

```
&&     Logical and
||     Logical or
!      Logical not.
```

Note: Use parentheses to limit operators if needed.

Include Command

This command tells `CreateReport` to find the file you specify and include it in the current text file. The full possible syntax of an include command is the following:

```
#include=filename[ ,<class>[.prop]]{[, [+,-]index]}#
```

The filename can be followed with a class (or object) name between `<>` characters. In this case, the file will be included once for each of the subobjects of the class (or object) and each occurrence of `!SELF!` in the included file will be substituted with that subobject name. Additionally, if the class (or object) name is followed with a property name, occurrences of `!PROP!` in the included file will be substituted with that property name.

Any property specified after `<class>[.prop]` is used for determining how the different subobjects should be sorted. Several of these properties can be used in which case the subobjects are first sorted on the first index, using the second index in case of a tie and so on. If the index is prefixed with a '-' (minus) character, the sorting is done in descending order. By default the order is ascending.

Result

The result returned by `CreateReport` is `TRUE` if the call is successful, `FALSE` if an error occurs.

Examples

A condition or action of the form

```
Execute "CreateReport" @STRING="@FILE=myfile";
```

will convert the file `myfile` and display the converted file.

If myfile contains the following lines:

```
#center#
Example of CreateReport file
#leftalign#
#if(\displayall\==TRUE)#
#include=myfile2.txt,<class>#
#endif#
End of CreateReport file
```

and myfile2 contains:

```
Object of class: !SELF! with value @V(!SELF!.Info)
```

and displayall is TRUE and class has two subobjects obj1 and obj2 with their property Info being Info1 and Info2, the converted file will be displayed as:

```
Example of CreateReport file
Object of class: obj1 with value Info1
Object of class: obj2 with value Info2
End of CreateReport file
```

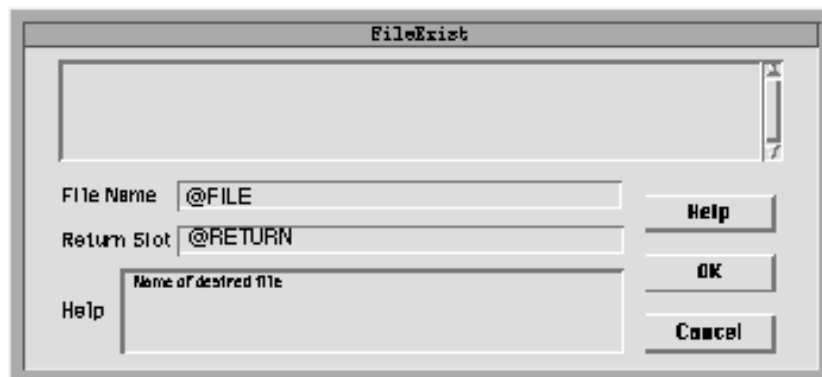
FileExist Routine

Definition

The Execute routine FileExist tests whether a designated file currently exists.

Interactive Dialog

FileExist is chosen with the Select Execute popup menu command in the Rule editor or Method editor, which automatically displays a special dialog box for specifying the execute parameters interactively, rather than by explicitly typing them in as listed below:



Parameters

The @ATOMID parameter is ignored.

The @STRING parameter must include the following:

@FILE=file_name Name of desired file.

@RETURN=answer_slot Name of slot in which to return result of test.

Both parameters are required. The destination specified by @RETURN must be a boolean-valued slot.

Effect

The destination slot designated by the @RETURN parameter is set to TRUE or FALSE, depending on whether the requested file currently exists. If the @FILE parameter does not specify a full path name, the file is sought in the current search path.

Result

The result returned by FileExist is TRUE if the call is successful, FALSE if an error occurs.

Examples

A condition or action of the form

```
Execute "FileExist"@STRING="@FILE=Flapdoo.dle,
        @RETURN=TheAnswer.Value";
```

will set TheAnswer.Value to TRUE if file Flapdoo.dle exists in the current search path, FALSE if it does not.

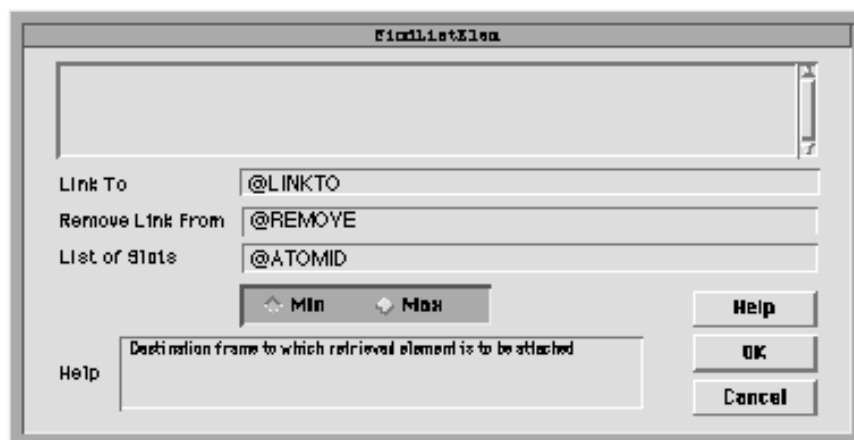
FindListElem Routine

Definition

The Execute routine FindListElem finds the largest or smallest object in a list according to the value of a designated property, and attaches it to a specified frame (object or class).

Interactive Dialog

FindListElem is chosen with the Select Execute popup menu command in the Rule editor or Method editor, which automatically displays a special dialog box for specifying the execute parameters interactively, rather than by explicitly typing them in as listed below:

**Parameters**

The @ATOMID parameter is a pattern specifying a list of slots (object properties) whose values are to be searched.

The @STRING parameter may include the following:

- @LINKTO=*destination* Destination frame to which retrieved element is to be attached.
- @MIN (Optional) Find smallest value in list.
- @MAX (Optional) Find largest value in list.
- @REMOVE=*parent_frame* (Optional) Parent frame from which element is to be detached.

Exactly one of the parameters @MIN and @MAX must be specified. The @REMOVE parameter, if present, must explicitly name a parent frame because the elements may have more than one parent.

Effect

The list of slots specified by @ATOMID is searched for the minimum or maximum value, as requested. The corresponding object is then attached to the frame named by the @LINKTO parameter as an instance or component (subobject). If a @REMOVE parameter is specified, the object is detached from the designated frame after being attached to the @LINKTO frame.

Result

The result returned by FindListElem is TRUE if the call is successful, FALSE if an error occurs.

Examples

Suppose the object Nephews has three components (subobjects) with the following properties:

```
nephew1.name = "Huey"      nephew1.capColor = "red"
nephew2.name = "Dewey"    nephew2.capColor = "green"
nephew3.name = "Louie"    nephew3.capColor = "blue"
```

Then a condition or action of the form

```
Execute "FindListElem" @ATOMID=<Nephews>.name;
                        @STRING="MIN,@LINKTO=SomeDucks";
```

will attach nephew2 (the object with the smallest value for property name) as a component of the object SomeDucks, while

```
Execute "FindListElem" @ATOMID=<Nephews>.capColor;
                        @STRING="MAX,@LINKTO=SomeDucks,
                        @REMOVE=Nephew";
```

will instead attach nephew1 (the object with the largest value for property capColor) as a component of SomeDucks, and will also remove it as an instance of class Nephew.

Note: The value types of the slots can be anything (STRING, INTEGER, TIME, DATE, etc.) and they will be compared accordingly. You don't need to specify the value types, however all the slots in the @ATOMID pattern must be the same type.

Related Topics

Execute Operator
Patterns

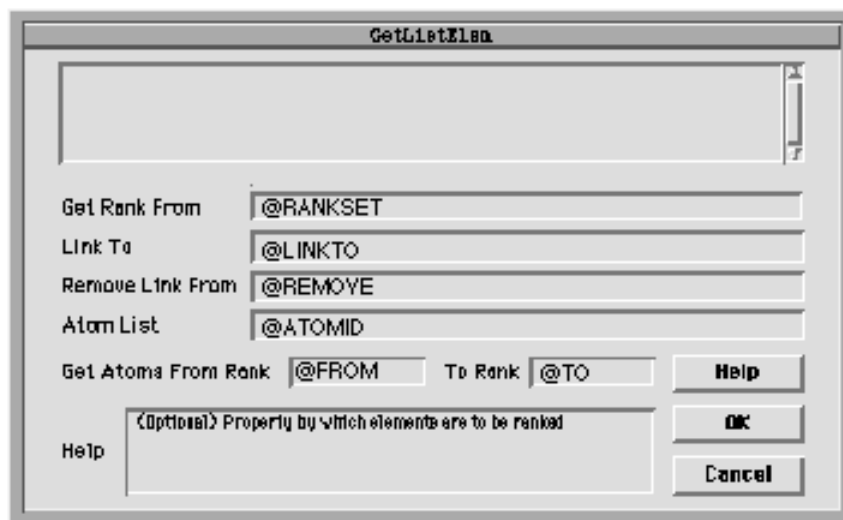
GetListElem Routine

Definition

The Execute routine `GetListElem` retrieves elements from a list of frames (objects or classes) and attaches them to another frame.

Interactive Dialog

`GetListElem` is chosen with the Select Execute popup menu command in the Rule editor or Method editor, which automatically displays a special dialog box for specifying the execute parameters interactively, rather than by explicitly typing them in as listed below:



Parameters

The `@ATOMID` parameter is a pattern specifying a list of frames (objects or classes) whose elements are to be retrieved.

The `@STRING` parameter may include the following:

`@LINKTO=destination` Destination frame to which retrieved elements are to be attached.

`@FROM=start_index` Index of first element to be retrieved. If *start_index* is negative, then counting starts from the end of the list.

`@TO=end_index` (Optional) Index of last element to be retrieved. If *end_index* is negative, then counting starts from the end of the list.

`@RANKSET=rank_prop` (Optional) Property by which elements are to be ranked. Property must be the type INT.

`@REMOVE=parent_frame` (Optional) Parent frame from which elements are to be detached.

The `@REMOVE` parameter, if present, must explicitly name a parent frame because the elements may have more than one parent.

Effect

The elements found at the given indices in the list specified by @ATOMID are attached as instances or components (subobjects) of the destination frame designated by @LINKTO. If no @TO index is given, only the single element at index @FROM is retrieved.

If either the @FROM or @TO parameters is negative, the counting starts from the end of the list. For example, @FROM = -1, @TO = -3 will get the last element through the third-from-last.

If a @RANKSET parameter is present, it identifies an integer property giving each list element's ordinal rank according to some ranking criterion (presumably assigned via an earlier call to the Execute routine RankList). The @FROM and @TO indices then refer to this logical rank rather than to the element's physical position within the list.

If a @REMOVE parameter is specified, the list elements are detached from the designated frame after being attached to the @LINKTO frame.

Result

The result returned by GetListElem is TRUE if the call is successful, FALSE if an error occurs.

Examples

Suppose class Duck has five instances whose name properties are equal to Donald, Daisy, Huey, Dewey, and Louie. Then a condition or action of the form

```
Execute "GetListElem"
@ATOMID=<Duck> ;@STRING="@FROM=2,@TO=4,
                @LINKTO=SomeDucks" ;
```

will attach Daisy, Huey, and Dewey (the second through fourth elements of the list) as components of the object SomeDucks.

If a previous call to RankList has ranked the instances of Duck alphabetically according to their name properties, setting

```
Daisy.name_rank = 1
Dewey.name_rank = 2
Donald.name_rank = 3
Huey.name_rank = 4
Louie.name_rank = 5
```

then

```
Execute "GetListElem"
@ATOMID=<Duck> ;@STRING="@FROM=2,@TO=4,
                @LINKTO=SomeDucks,@RANKSET=name_rank,
                @REMOVE=Duck" ;
```

will instead attach Dewey, Donald, and Huey (the second- through fourth-ranked elements according to property name_rank) as components of SomeDucks, and will also remove them as instances of class Duck.

If you used the parameters in the above example, @FROM=-1, @TO=-2, the last through second to last elements, namely HUEY and LOUIE, are attached to SomeDucks since the indices are negative.

Related Topics

Execute Operator

Patterns

RankList Routine

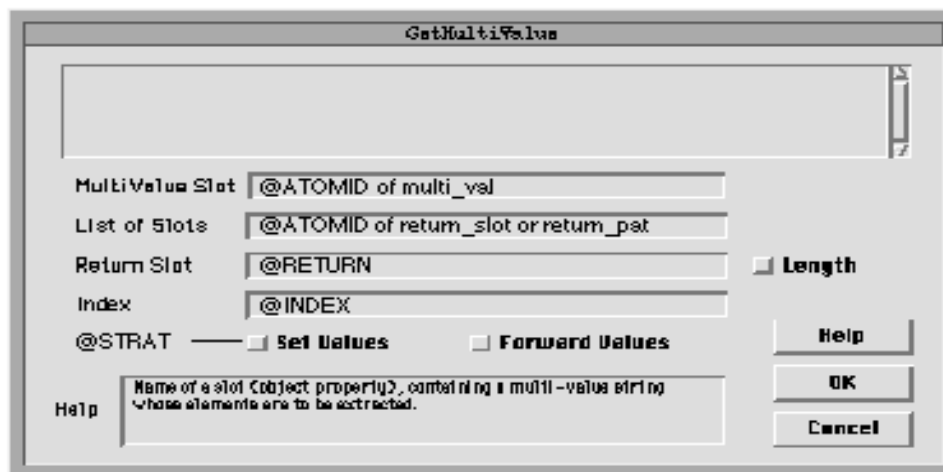
GetMultiValue Routine

Definition

The Execute routine `GetMultiValue` extracts one or more elements from a multi-value.

Interactive Dialog

`GetMultiValue` is chosen with the Select Execute popup menu command in the Rule editor or Method editor, which automatically displays a special dialog box for specifying the execute parameters interactively, rather than by explicitly typing them in as listed below:



Parameters

The `@ATOMID` parameter consists of one or two items:

- The name of a slot (object property), `multi_val`, containing a multi-value string whose elements are to be extracted
- (Optional) A slot name, `return_slot`, or a pattern, `return_pat`, specifying a list of slots to receive the extracted elements

The `@STRING` parameter may include the following:

`@INDEX=index_number` (Optional) Index of desired element.

`@LENGTH` (Optional) Requests number of elements in multi-value.

`@RETURN=length_slot` (Optional) Slot in which to return number of elements.

@STRAT=*options* (Optional) Strategy options governing the assignment (see Execute Library Overview for details).

If **@LENGTH** is specified, then **@RETURN** must be included as well.

Effect

If an **@INDEX** parameter is given, the element at that index in `multi_val` is returned as the value of `return_slot`. If the specified index exceeds the number of elements in the multi-value, a warning will be posted to the transcript, but the Execute routine itself will not fail.

If no **@INDEX** parameter is given, all elements of `multi_val` are extracted and assigned individually to the slots designated by `return_pat`.

If **@LENGTH** is specified, the length of `multi_val` (the number of elements it contains) is assigned as the value of `length_slot`.

Result

The result returned by `GetMultiValue` is **TRUE** if the call is successful, **FALSE** if an error occurs.

Examples

Suppose the object `Nephews` has three components (subobjects) named `Huey`, `Dewey`, and `Louie`, each of which has a property named `capColor`. If `TheColors.Value` contains the multi-value `red, green, blue`, then

```
Execute "GetMultiValue" @ATOMID=TheColors.Value,
                    <Nephews>.capColor;
```

will assign `red, green, and blue` to `Huey.capColor`, `Dewey.capColor`, and `Louie.capColor`, respectively.

```
Execute "GetMultiValue"
@ATOMID=TheColors.Value, Dewey.capColor;
@STRING="@INDEX=2";
```

will set `Dewey.capColor` to `green`, the second element of `TheColors.Value`, and

```
Execute "GetMultiValue" @ATOMID=TheColors.Value;
@STRING="@LENGTH,
@RETURN=TheColors.len";
```

will set `TheColors.len` to `3`, the number of elements in `TheColors.Value`.

Note: If the number of elements in the multivalued does not match the number of slots in the `return_pat`, a warning will be posted in the transcript, but the Execute routine itself will not fail.

Related Topics

Execute Operator

Multi-Values

Patterns

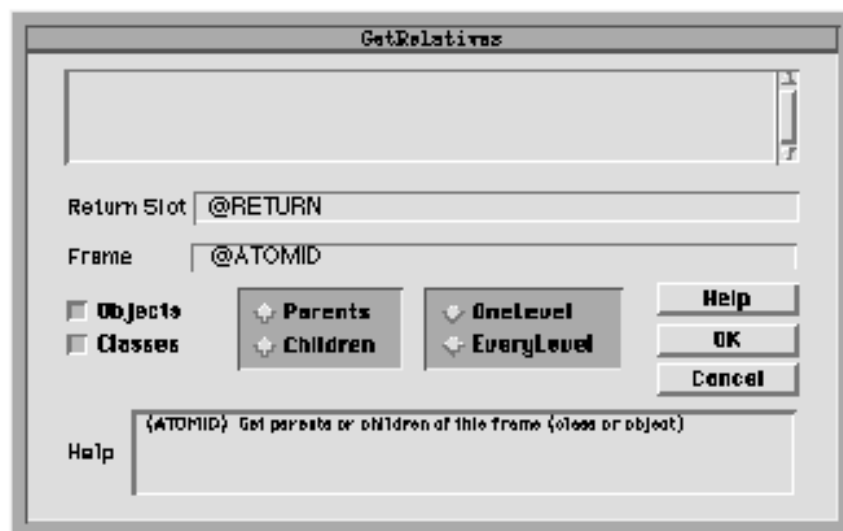
GetRelatives Routine

Definition

The Execute routine `GetRelatives` stores the inheritance pathway class and/or object names of a given frame in a string slot as a multi-value.

Interactive Dialog

`GetRelatives` is chosen with the Select Execute popup menu command in the Rule editor or Method editor, which automatically displays a special dialog box for specifying the execute parameters interactively, rather than by explicitly typing them in as listed below:



Parameters

The `@ATOMID` parameter is the name of a frame whose inheritance pathway is to be tested.

The `@STRING` parameter may include the following:

<code>@ONELEVEL</code>	(Optional) Get only immediate parents or children.
<code>@EVERYLEVEL</code>	(Optional) Get all parents or children up or down from every level of inheritance.
<code>@CHILDREN</code>	(Optional) Get descendants for class or object.
<code>@PARENTS</code>	(Optional) Get ancestors for class or object.
<code>@CLASSES</code>	(Optional) Report relatives that are classes.
<code>@OBJECTS</code>	(Optional) Report relatives that are objects.
<code>@RETURN=<i>multi_val</i></code>	Name of slot in which to report results.

The parameters `@CHILDREN` and `@PARENTS` are mutually exclusive, and may not both be specified, as are `@ONELEVEL` and `@EVERYLEVEL`. If `@CLASSES` and `@OBJECTS` are omitted then both classes and objects are reported.

Effect

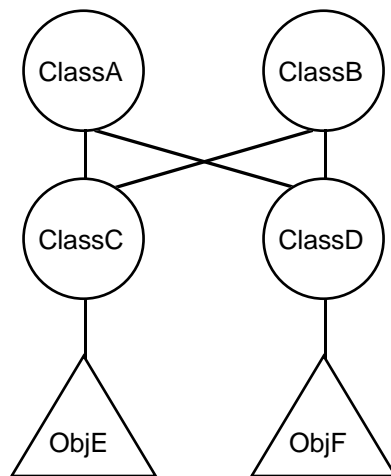
The target slot @RETURN contains the names of the relatives of the specified atom as a multi-value. Relatives are the objects and classes that form the inheritance pathways of the specified atom (@ATOMID). The relatives can be the parents or children, classes and/or objects, immediate or all inclusive depending on the @STRING options specified.

Result

The result returned by `GetRelative` is `TRUE` if the call is successful, `FALSE` if an error occurs.

Example

`ClassA` and `ClassB` both have children `ClassC` and `ClassD`. `ClassC` has a child `ObjE` and `ClassD` has a child `ObjF` like this:



```
GetRelatives (@STRING="@ONELEVEL, @CHILDREN,
              @RETURN=answer.mulVal" ; @ATOMID=ClassA; )
```

This will return the multi-value `ClassC, ClassD` in `answer.mulVal`.

```
GetRelatives (@STRING="@EVERYLEVEL, @CHILDREN,
              @RETURN=answer.mulVal" ; @ATOMID=ClassB; )
```

This will return the multi-value `ClassC, ClassD, ObjE, ObjF` in `answer.mulVal`.

```
GetRelatives (@STRING="@EVERYLEVEL, @CHILDREN, @CLASSES,
              @RETURN=answer.mulVal" ; @ATOMID=ClassB; )
```

This will return the multi-value `ClassC, ClassD` in `answer.mulVal`.

Notice that `ObjE` and `ObjF` are not included because we specified `@CLASSES` only.

```
GetRelatives (@STRING="@EVERYLEVEL,
              @PARENTS, @RETURN=answer.multiVal" ;
              @ATOMID=ObjE; )
```

This will return the multi-value `ClassC, ClassA, ClassB` in slot `answer.multiVal`.

Journal Routine

Definition

The Execute routine `Journal` performs all of the Rules Element's standard journaling operations.

Interactive Dialog

`Journal` is chosen with the Select Execute popup menu command in the Rule editor or Method editor, which automatically displays a special dialog box for specifying the execute parameters interactively, rather than by explicitly typing them in as listed below:



Parameters

The `@ATOMID` parameter is ignored.

The `@STRING` parameter may include the following:

<code>@operation</code>	Journaling operation to be performed (see Operations below).
<code>@FILE=<i>file_name</i></code>	(Optional) Name of journal file.
<code>@PLAYSTEP</code>	(Optional) Replay step by step.
<code>@PLAYSKIPSHOW</code>	(Optional) Skip Show operators.
<code>@PLAYNOSCAN</code>	(Optional) Don't scan file.

The `@FILE` parameter is not needed with the journaling operations `@RECORDSTOP` and `@PLAYSTOP` (see "Operations," below), but is required with all other operations. The last three parameters are meaningful only in connection with the `@PLAYSTART` operation.

Operations

The operation specifier included in the `@STRING` parameter identifies the journaling operation to be performed. It must consist of exactly one of the following:

<code>@RECORDSTART</code>	Start Recording
<code>@RECORDSTOP</code>	Stop recording
<code>@PLAYSTART</code>	Start playback

@PLAYSTOP	Stop playback
@VALUESSAVE	Save slot values only
@STATESAVE	Save complete state
@STATERESTORE	Restore complete state

All operations except @RECORDSTOP and @PLAYSTOP require a @FILE parameter to identify the journal file to be used. The @PLAYSTART operation may optionally be modified by including the additional parameters @PLAYSTEP, @PLAYNOSCAN, or @PLAYSKIPSHOW.

Effect

The journaling operation specified in the @STRING parameter is executed.

Result

The result returned by Journal is TRUE if the call is successful, FALSE if an error occurs.

Examples

A condition or action of the form

```
Execute "Journal" @STRING="@STATESAVE,@FILE=Session.jou";
```

will save the current state of the session in the journal file Session.jou; thereafter,

```
Execute "Journal"
@STRING="@STATERESTORE,@FILE=Session.jou";
```

will restore the session to the state previously saved.

Related Topics

Execute Operator
Journaling

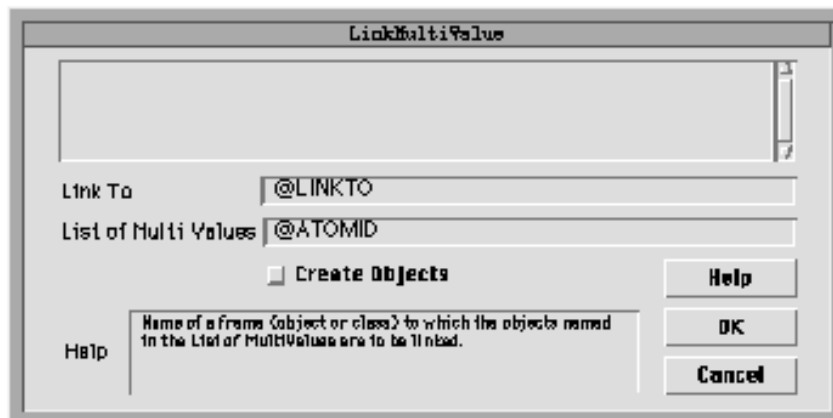
LinkMultiValue Routine

Definition

The Execute routine LinkMultiValue creates links from objects listed by name in a list of multi-values to a specified class or object.

Interactive Dialog

`LinkMultiValue` is chosen with the Select Execute popup menu command in the Rule editor or Method editor, which automatically displays a special dialog box for specifying the execute parameters interactively, rather than by explicitly typing them in as listed below:



Parameters

The `@ATOMID` parameter is a list of multivalue slots which contain the names of objects to be linked.

The `@STRING` parameter may include the following:

- `@LINKTO=atom_name` Objects named in the multi-values are linked to this frame (object or class).
- `@CREATEOBJECTS` (Optional) Ensures all objects named in the multi-values are linked whether they already exist or not.

If you omit the `@CREATEOBJECTS` parameter, you must ensure the names in the multivalues are legitimate object names.

Effect

The values in the multivalue lists become objects linked to the specified object or class. If the `@CREATEOBJECTS` parameter is specified, new objects are created; otherwise, the names in the multivalue lists must already exist in the system as object names.

Result

The result returned by `LinkMultiValue` is `TRUE` if the call is successful, `FALSE` if an error occurs.

Examples

Assume `<MClass>` contains three objects `Obj1`, `Obj2`, and `Obj3` with a string property `mval` for holding a multi-value. The current values are as follows:

```
Obj1.mval = "alpha,beta,charlie"
Obj2.mval = "delta,echo,fox"
Obj3.mval = "gulf,hotel,india"
```

Assume that the objects alpha, beta, charlie, delta, and echo already exist.

```
LinkMultiValue (@STRING="@LINKTO=myFrame"; @ATOMID=Obj1.mval;)
```

This will link all of the objects whose names are in Obj1.mval to the frame myFrame. So, alpha, beta, and charlie will all be linked to myFrame.

```
LinkMultiValue (@STRING="@LINKTO=myFrame";
                @ATOMID=Obj1.mval,Obj2.mval;)
```

This will link the objects in Obj1.mval and Obj2.mval to myFrame. However, since fox does not exist, it will not be created or linked.

```
LinkMultiValue (@STRING="@LINKTO=myFrame, @CREATEOBJECTS";
                @ATOMID=<MClass>.mval;)
```

This will link all objects whose names are in all of the multi-values that are in the class MClass to the frame myFrame. Since @CREATEOBJECTS is specified, the objects that don't exist yet (fox, gulf, hotel and india) will be created.

Related Topics

Inheritance

Multi-Values

Execute Operator

AtomName Routine

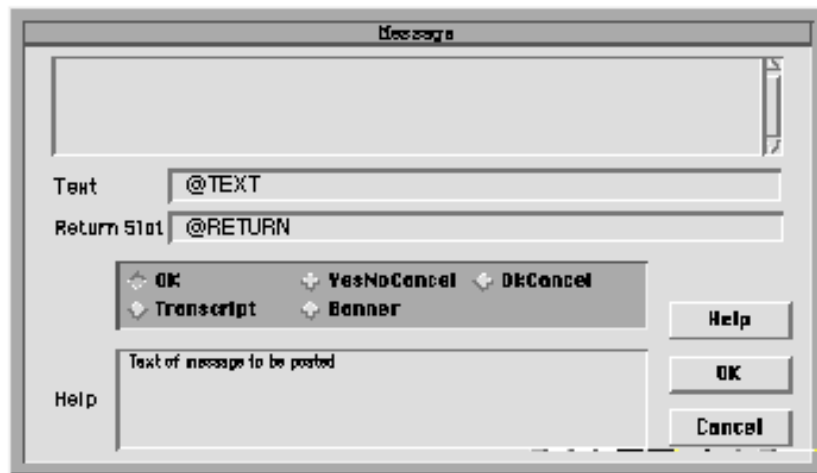
Message Routine

Definition

The Execute routine *Message* posts a message on the screen or sends one to the banner or transcript handler.

Interactive Dialog

Message is chosen with the Select Execute popup menu command in the Rule editor or Method editor, which automatically displays a special dialog box for specifying the execute parameters interactively, rather than by explicitly typing them in as listed below:



Parameters

The @ATOMID parameter is ignored.

The @STRING parameter may include the following:

@TEXT= <i>text_string</i>	Text of message to be posted.
@RETURN= <i>reply_slot</i>	(Optional) Name of slot in which to return user's reply.
@OK	(Optional) If present, use dialog box with one button labeled OK .
@OKCANCEL	(Optional) If present, use dialog box with two buttons labeled OK and Cancel .
@YESNOCANCEL	(Optional) If present, use dialog box with three buttons labeled Yes, No, and Cancel .
@BANNER	(Optional) If present, send message to banner handler.
@TRANSCRIPT	(Optional) If present, send message to transcript handler.
@STRAT= <i>options</i>	(Optional) Strategy options governing assignment to reply slot (see Execute Library Overview for details).

The parameters @OK, @OKCANCEL, @YESNOCANCEL, @BANNER, and @TRANSCRIPT are mutually exclusive; at most one may be specified. If none is present, @OK is assumed by default.

The @RETURN parameter is needed only with @OKCANCEL or @YESNOCANCEL, and will be ignored if @OK, @BANNER, or @TRANSCRIPT is specified.

Effect

If @OK, @OKCANCEL, or @YESNOCANCEL is specified, the message given by the @TEXT parameter is displayed in a dialog box with the requested number of buttons using the Alert Handler. The value returned in the @RETURN parameter identifies the button the user used to dismiss the dialog:

1	OK or Yes
0	Cancel
-1	No

If @BANNER or @TRANSCRIPT is specified, the message is sent to the banner or transcript handler instead of an on-screen dialog box; no result value is returned.

Result

The result returned by Message is TRUE if the call is successful, FALSE if an error occurs.

Examples

A condition or action of the form

```
Execute "Message" @STRING="@TEXT=Do you want to continue?,
@OKCANCEL,@RETURN=answer.Value";
```

will post the message `Do you want to continue?` in a dialog box with two buttons labeled `OK` and `Cancel`. The contents of `answer.Value` will be set to 1 or 0 to indicate whether the user clicked `OK` or `Cancel`.

A condition or action of the form

```
Execute "Message" @STRING="@TEXT=Now entering rule 5,
@TRANSCRIPT";
```

will post the message `Now entering rule 5` to the transcript.

Related Topics

Execute Operator

Multi-Values

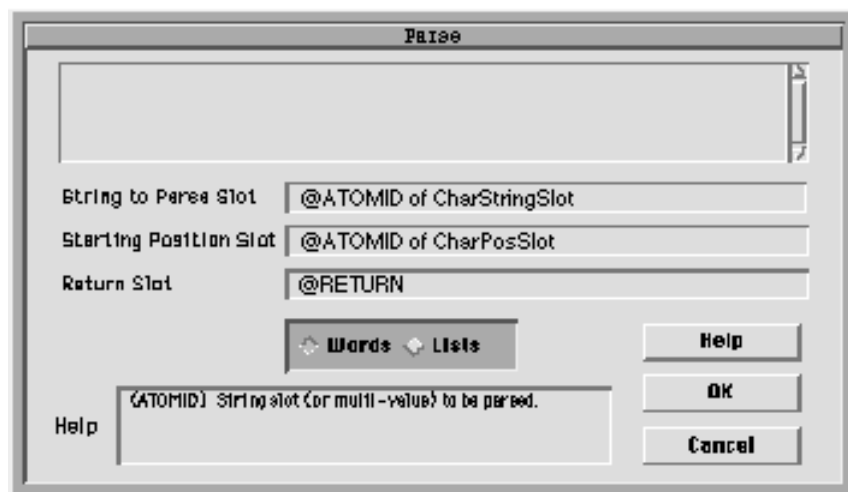
Parse Routine

Definition

The `Execute` routine `Parse` separates a larger string into its component parts and stores the next string token into a slot.

Interactive Dialog

`Parse` is chosen with the `Select Execute` popup menu command in the `Rule` editor or `Method` editor, which automatically displays a special dialog box for specifying the execute parameters interactively, rather than by explicitly typing them in as listed below:



Parameters

The @ATOMID parameter consists of two parts:

- CharPosSlot: The character position from which to begin parsing is an integer slot.
- CharStringSlot: The slot that contains the string to be parsed into component strings.

The @STRING parameter may include the following:

- @WORDS (Optional) Search for string tokens separated by spaces.
- @LIST (Optional) Search for string tokens separated by commas.
- @RETURN=*string_slot* Name of slot in which to return the next token after the current character position.

If @WORDS and @LIST are omitted, the system uses the default @WORDS mode. The destination specified by @RETURN must be a string slot.

Effect

This execute parses the @ATOMID string slot for either words or list elements. In either @WORDS or @LIST mode, the ParseStringSlot will be parsed starting from the character position in the integer slot CharPosSlot. The next string token found will be returned in the StringSlot. If no token is found, an empty string will be returned, and the CharPosSlot will be set to -1. If a token is found, CharPosSlot will be advanced to the next character position after the token. This enables you to set up a looping rule which parses out each token one by one. (See examples.)

In @WORDS mode, a token is defined as a string of visible (non-blank) characters separated by spaces. In this mode, a comma or an equals sign can also separate two tokens. In that case, the comma or equals sign would be considered as a separate token. For example, the following shows how a string would be parsed in @WORDS mode:

```
The string: "hello there a=b 1,2"
Token 1: "hello"
Token 2: "there"
Token 3: "a"
Token 4: "="
Token 5: "b"
Token 6: "1"
Token 7: ","
Token 8: "2"
```

In @LIST mode, a token is defined as a string of characters separated by commas. In this case, the commas are not considered tokens, just separators. The leading and trailing blanks in a token are eliminated, but embedded blanks are retained. Here is an example of parsing in @LIST mode:

```
The string: "item1, item2, two words"
Token 1: "item1"
Token 2: "item2"
Token 3: "two words"
```

Result

The result returned by `Parse` is `TRUE` if the call is successful, `FALSE` if an error occurs.

Examples

The following example shows how to set up 2 rules which parse the words out of a sentence:

Example 1

Rule 1:

```
If there is evidence of something
    And "This is a sentence" is assigned to
ParseString.strVal
    And 0 is assigned to CharPos.intVal
    And there is no evidence of ParseLoop
Then Hypo
    is confirmed.
```

Rule 2:

```
If CharPos.intVal is greater than or equal to 0
    And Execute "Parse" (@ATOMID=CharPos.intVal,
        ParseString.strVal"; @STRING="@WORDS,
        @RETURN=Token.strVal");
    And <...do something with Token.strVal here...>
Then ParseLoop
    is confirmed.
    And Reset ParseLoop
```

When `Hypo` is suggested, the string `This is a sentence` is assigned to `ParseString.strVal`, and `CharPos.intVal` is set to zero. Then, the next condition forces backward chaining to rule 2. That rule checks to see if `CharPos.intVal` is greater than or equal to zero. Since it is, it then executes `Parse`. `Parse` will return the first token in `ParseString.strVal` by setting `Token.strVal` to `This`. You can then do whatever you want with that token. `Parse` also sets `CharPos.intVal` to the character position right after the token, which in this case would be 4 (since the count starts at 0). On the right hand side of the rule, the hypothesis `ParseLoop` is reset which causes it to be executed again.

The next time Rule 2 is executed, `CharPos.intVal` will be 4, so the token returned by `Parse` will be `is`. The loop continues in this manner until no more tokens are found. At that point, `CharPos.intVal` is set to -1, and the hypothesis `ParseLoop` is rejected which then causes `Hypo` to be confirmed.

Examples 2 and 3

The following two examples show the difference between parsing in `@WORDS` mode and `@LIST` mode. For both examples, `CharPos.intVal` contains 0 and `ParseString.strVal` contains the following string:

```
"Hello there, Bob"
@PARSE (@STRING="@WORDS, @RETURN=Token.strVal";
        @ATOMID=CharPos.intVal, ParseString.strVal);
```

After executing this, `Token.strVal` will contain `Hello`, and `CharPos.intVal` will contain 5 since the token ends on character 4 (starting the count with 0). If this were executed again, the next time `Token.strVal` would contain `there` and `CharPos.intVal` would contain 11. The next time, `Token.strVal` would contain `,` and

`CharPos.intVal` would contain 12. Then, Bob and 16. Finally, on the fifth try, the token would be empty, and `CharPos.intVal` would be set to -1 to indicate that there are no more tokens in the string.

```
@PARSE (@STRING="@LIST, @RETURN=Token.strVal";
        @ATOMID=CharPos.intVal, ParseString.strVal;)
```

After executing this, `Token.strVal` will contain Hello there and `CharPos.intVal` will contain 11. Since we are in `@LIST` mode, everything up to the comma is considered part of the token except for leading and trailing blanks. If we execute this again, `Token.strVal` will contain Bob and `CharPos.intVal` will contain 16. Finally, a third execution will cause the token to be empty and `CharPos.intVal` will be -1. Notice in `@LIST` mode, the comma was never returned as a token. `@LIST` mode is useful for parsing lists such as multi-values.

Related Topics

Execute Operator

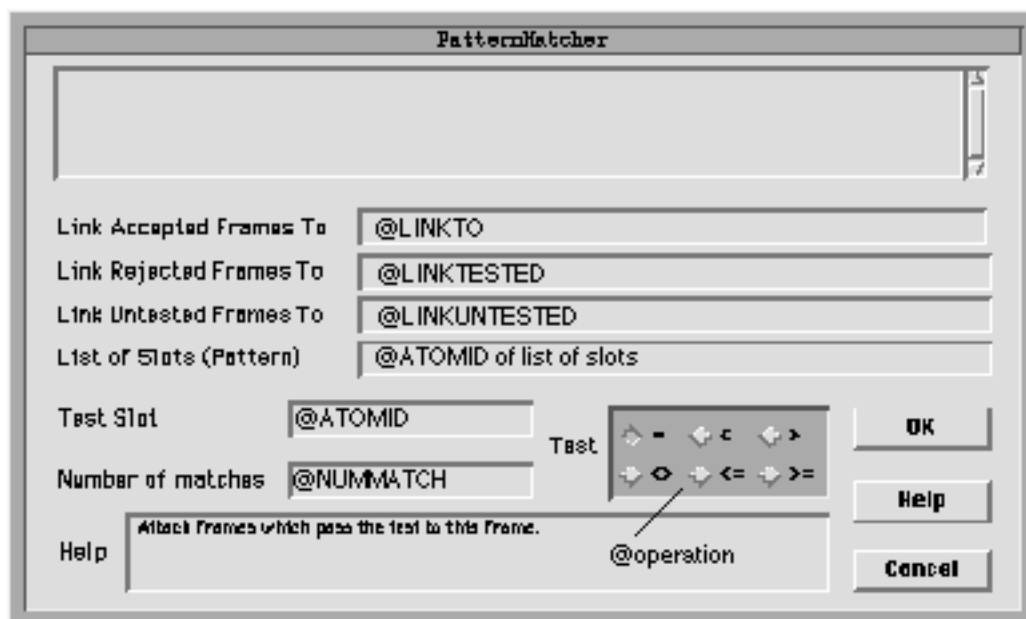
PatternMatcher Routine

Definition

The Execute routine `PatternMatcher` compares a slot against a list of slots and links a specified number of matches to a specified class.

Interactive Dialog

`PatternMatcher` is chosen with the Select Execute popup menu command in the Rule editor or Method editor, which automatically displays a special dialog box for specifying the execute parameters interactively, rather than by explicitly typing them in as listed below:



Parameters

The @ATOMID parameter consists of two parts:

- The test slot that you want to compare to the list of slots.
- The list of slots that you perform the test on.

The @STRING parameter may include the following:

- @LINKTO=*destination* Destination frame to which successfully tested elements are attached.
- @LINKTESTED=*testedFrame*
(Optional) Frame to which tested but failed elements are attached.
- @LINKUNTESTED=*untestedFrame*
(Optional) Frame to which not yet tested elements are attached.
- @NUMMATCH=*number* (Optional) Continue test until specified number of matches are found.
- @operation (Optional) Test operation to be performed (see Operations below).

If @NUMMATCH is omitted, the system uses the default 1. Also if no test operator keyword is supplied, the default is @EQUAL.

Operations

The operation specifier included in the @STRING parameter identifies the operation to be performed on the pair of multi-values designated by @ATOMID and @VALUE. It must consist of exactly one of the following:

- @EQUAL All elements in ListOfSlots that have values equal to the value of testSlot
- @NOT_EQUAL All elements in ListOfSlots that have values that are not equal to the value of testSlot
- @LESS All elements in ListOfSlots that have values less than the value of testSlot
- @LESS_EQUAL All elements in ListOfSlots that have values less than or equal to the value of testSlot
- @GREATER All elements in ListOfSlots that have values greater than the value of testSlot
- @GREATER_EQUAL All elements in ListOfSlots that have values greater than or equal to the value of testSlot

Effect

The PatternMatcher tests each of the slots in the ListOfSlots against the testSlot according to one of the test operators. As soon as the specified number of matches is found, PatternMatcher stops checking. For example, if the ListOfSlots has five slots that pass the test, but @NUMMATCH was set to 3, only the first three successful tests will be linked to the linkFrame.

Optionally, you can also have all slots which were tested but don't pass the test condition attached to the `testedFrame`, and all slots which have yet to be tested linked to the `untestedFrame`. If the actual number of matches is less than the number specified in `@NUMMATCH`, then the whole list will be searched and nothing will be linked to the `untestedFrame`.

Result

The result returned by `PatternMatcher` is `TRUE` if the call is successful, `FALSE` if an error occurs.

Examples

Assume `<Class1>` has two properties `weight` and `color` and five objects `Obj1` through `Obj5`. The current values are as follows:

```
Obj1.weight = 10   Obj1.color = blue
Obj2.weight = 20   Obj2.color = green
Obj3.weight = 30   Obj3.color = orange
Obj4.weight = 40   Obj4.color = red
Obj5.weight = 50   Obj5.color = yellow
```

Another object `tester` also has two properties `pounds` and `finish` with values as follows:

```
tester.pounds = 25  tester.finish = "orange"
```

Also, we have three classes for attaching results: `linkClass`, `testedClass`, and `untestedClass`.

```
PatternMatcher (@STRING="@LINKTO=linkClass, @GREATER,
@NUMMATCH=2";
                @ATOMID=tester.pounds, <Class1>.weight; )
```

This will match the first two objects in `Class1` whose `weight` is greater than `tester.pounds`. So, in this case, `Obj3` and `Obj4` will be linked to `linkClass`.

```
PatternMatcher (@STRING="@LINKTO=linkClass,
@LINKTESTED=testedClass,
@LINKUNTESTED=untestedClass,
@GREATER, @NUMMATCH=2";
                @ATOMID=tester.pounds, <Class1> weight; )
```

This is basically the same as the previous example, except that we are linking the tested and untested objects to frames. So, in this case, `Obj1` and `Obj2` will be linked to `testedClass`, `Obj3` and `Obj4` will be linked to `linkClass`, and `Obj5` will be linked to `untestedClass`.

```
PatternMatcher (@STRING="@LINKTO=linkClass, @EQUAL";
                @ATOMID=tester.finish, <Class1>.color; )
```

This will find the first object in `Class1` whose `color` is equal to the `finish` in `tester`. So, in this case, `Obj3` will be linked to `linkClass`.

Related Topics

Patterns

Execute Operator

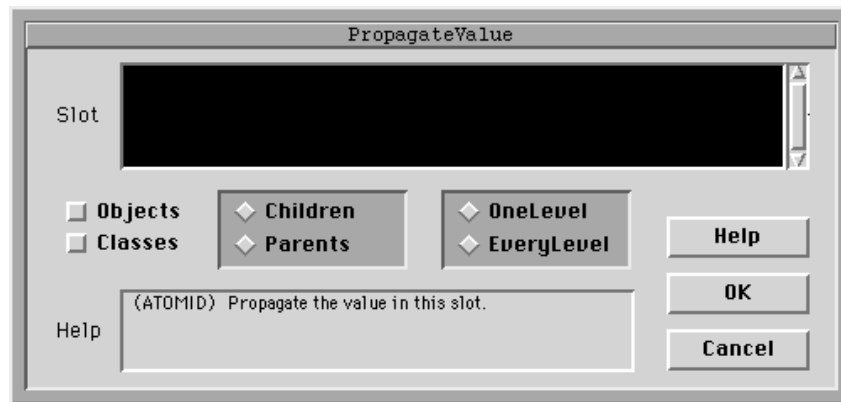
PropagateValue Routine

Definition

The Execute routine `PropagateValue` assigns the value of a specified atom to atoms in the inheritance pathway that contain the same property.

Interactive Dialog

`PropagateValue` is chosen with the Select Execute popup menu command in the Rule editor or Method editor, which automatically displays a special dialog box for specifying the execute parameters interactively, rather than by explicitly typing them in as listed below:



Parameters

The `@ATOMID` parameter is the name of a slot whose properties you wish to propagate.

The `@STRING` parameter may include the following:

<code>@ONELEVEL</code>	(Optional) Propagate only to immediate parents or children.
<code>@EVERYLEVEL</code>	(Optional) Propagate to all parents or children up or down from <code>@ATOMID</code> .
<code>@CHILDREN</code>	(Optional) Propagate down to descendants.
<code>@PARENTS</code>	(Optional) Propagate up to ancestors.
<code>@CLASSES</code>	(Optional) Propagate to relatives that are classes.
<code>@OBJECTS</code>	(Optional) Propagate to relatives that are objects.

The parameters `@CHILDREN` and `@PARENTS` are mutually exclusive, and may not both be specified, as are `@ONELEVEL` and `@EVERYLEVEL`. If `@CLASSES` and `@OBJECTS` are omitted then both classes and objects are used.

Effect

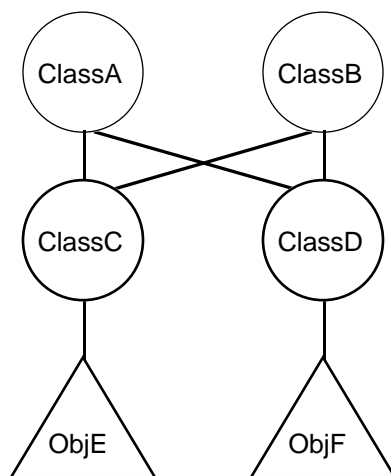
The relatives that share the same property as the specified atom receive the value of that atom. Relatives are the objects and classes that form the inheritance pathways of the specified atom. The relatives can be the parents or children, classes and/or objects, immediate or all inclusive depending on the @STRING options specified.

Result

The result returned by `PropagateValue` is `TRUE` if the call is successful, `FALSE` if an error occurs.

Examples

A `<ClassA>` and `<ClassB>` both have subclasses `<ClassC>` and `<ClassD>`. `<ClassC>` has a subobject `ObjE`. `<ClassD>` has a subobject `ObjF`:



```
PropagateValue (@STRING="@EVERYLEVEL, @CHILDREN"; @ATOMID=ClassA.intval;)
```

This will propagate the current value in `ClassA.intval` to all of the children which have a property `intval` on all levels. So, whatever the current value is `ClassA.intval` is, that value will be propagated to `ClassC.intval`, `ClassD.intval`, `ObjE.intval` and `ObjF.intval`.

```
PropagateValue (@STRING="@ONELEVEL, @PARENTS"; @ATOMID=ClassC.intval;)
```

This will propagate the current value of `ClassC.intval` to its parents, `ClassA.intval` and `ClassB.intval`. If those objects do not have the property `intval`, it will not be created and the value will not be propagated.

```
PropagateValue (@STRING="@EVERYLEVEL, @CHILDREN, @CLASSES";
                @ATOMID=ClassA.intval;)
```

This will propagate the current value of `ClassA.intval` to all of the children classes (not objects). So, the value will be propagated to `ClassC.intval` and `ClassD.intval`.

Related Topics

Patterns

Execute Operator

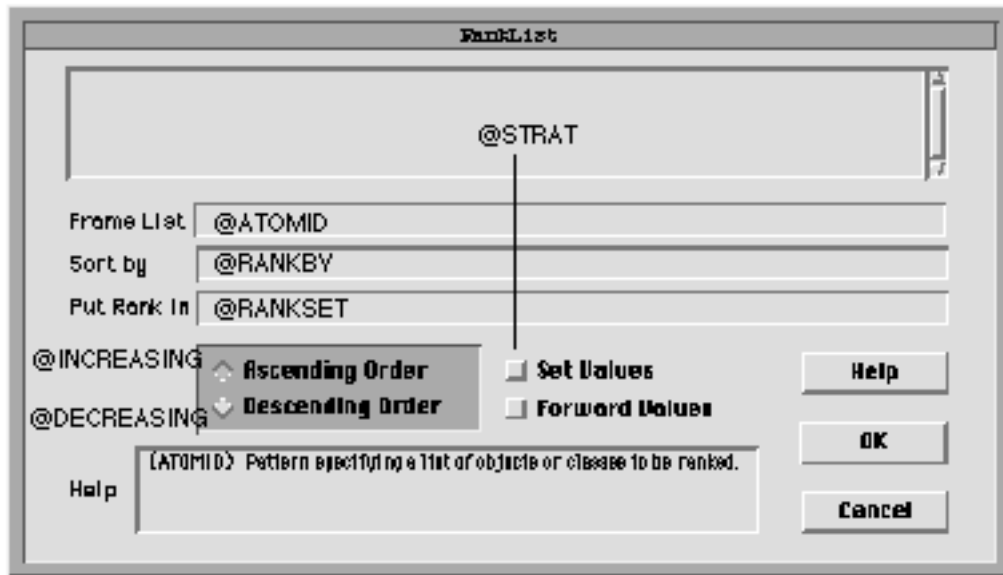
RankList Routine

Definition

The Execute routine `RankList` ranks a list of objects or classes according to the value of a designated property.

Interactive Dialog

`RankList` is chosen with the Select Execute popup menu command in the Rule editor or Method editor, which automatically displays a special dialog box for specifying the execute parameters interactively, rather than by explicitly typing them in as listed below:



Parameters

The `@ATOMID` parameter is a pattern specifying a list of objects or classes to be ranked.

The `@STRING` parameter may include the following:

- `@RANKBY=rank_prop` Property determining ranking.
- `@RANKSET=set_prop` Property into which rank is to be stored.
- `@INCREASING` (Optional) If present, rank in increasing order.
- `@DECREASING` (Optional) If present, rank in decreasing order.
- `@STRAT=options` (Optional) Strategy options governing the assignment (see Execute Library Overview for details).

At most one of the parameters `@INCREASING` and `@DECREASING` may be specified; if neither is present, `@INCREASING` is assumed by default.

Effect

The objects or classes specified by @ATOMID are ranked according to the value of the property designated by @RANKBY. The property designated by @RANKSET is then set to the corresponding numerical rank, from 1 to the length of the list.

If the @INCREASING parameter is specified, a rank of 1 denotes the object or class with the smallest value for the designated property; if @DECREASING, the one with the greatest value.

The rank_prop can be any type (STRING, INTEGER, TIME, DATE, etc.) but all of the objects or classes in the pattern must have this property. Also, the set_prop must be INTEGER type and all of the objects or properties in the pattern must have this property.

Result

The result returned by RankList is TRUE if the call is successful, FALSE if an error occurs.

Examples

If class Duck has five instances whose name properties are equal to Donald, Daisy, Huey, Dewey, and Louie, then

```
Execute "RankList" @ATOMID=<Duck>;@STRING= "@RANKBY=name,
                                     @RANKSET=name_rank,@INCREASING";
```

will rank the instances alphabetically by their name fields, setting Daisy.name_rank equal to 1, Dewey.name_rank to 2, Donald.name_rank to 3, Huey.name_rank to 4, and Louie.name_rank to 5, while

```
Execute "RankList" @ATOMID=<Nephews>;@STRING= "@RANKBY=name,
                                     @RANKSET=name_rank, @DECREASING";
```

will set Daisy.name_rank to 5, Dewey.name_rank to 4, Donald.name_rank to 3, Huey.name_rank to 2, and Louie.name_rank to 1.

Related Topics**Patterns****Execute Operator**

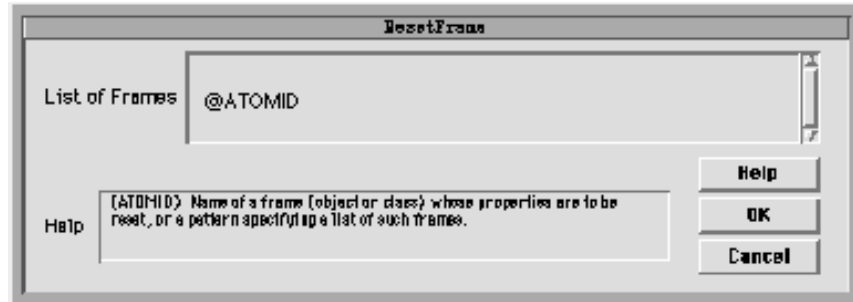
ResetFrame Routine

Definition

The Execute routine ResetFrame resets all properties of one or more frames (objects or classes) to UNKNOWN.

Interactive Dialog

`ResetFrame` is chosen with the Select Execute popup menu command in the Rule editor or Method editor, which automatically displays a special dialog box for specifying the execute parameters interactively, rather than by explicitly typing them in as listed below:



Parameters

The `@ATOMID` parameter is the name of a frame (object or class) whose properties are to be reset, or a pattern specifying a list of such frames.

The `@STRING` parameter is ignored.

Effect

All properties of each object or class designated by the `@ATOMID` parameter are reset to UNKNOWN.

Result

The result returned by `ResetFrame` is TRUE if the call is successful, FALSE if an error occurs.

Examples

Suppose class `Cube` has two instances named `cube1` and `cube2` and three properties named `width`, `height`, and `depth`. Then an action of the form

```
Execute "ResetFrame" @ATOMID=cube1;
```

will reset the properties

```
cube1.width
cube1.height
cube1.depth
```

to UNKNOWN, and

```
Execute "ResetFrame" @ATOMID=<Cube>;
```

will reset

```
cube1.width cube2.width
cube1.height cube2.height
cube1.depth cube2.depth
```

Related Topics

Data Types

Patterns

Execute Operator

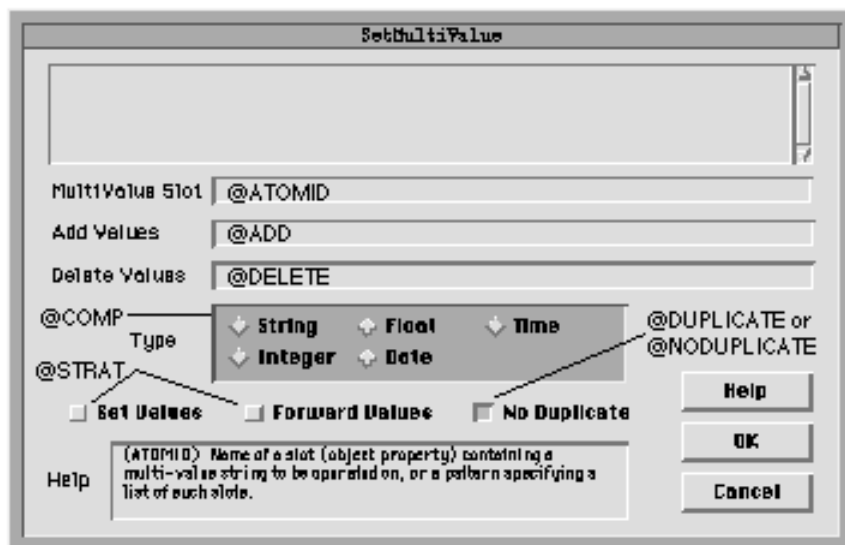
SetMultiValue Routine

Definition

The Execute routine `SetMultiValue` adds or deletes elements from one or more multi-values.

Interactive Dialog

`SetMultiValue` is chosen with the Select Execute popup menu command in the Rule editor or Method editor, which automatically displays a special dialog box for specifying the execute parameters interactively, rather than by explicitly typing them in as listed below



Parameters

The `@ATOMID` parameter is the name of a slot (object property) containing a multi-value string to be operated on, or a pattern specifying a list of such slots.

The `@STRING` parameter may include the following:

- `@ADD=value_list` (Optional) List of elements to be added.
- `@DELETE=value_list` (Optional) List of elements to be deleted.
- `@DUPLICATE` (Optional) Allow duplicate occurrences of the same element in a multi-value.
- `@NODUPLICATE` (Optional) Avoid duplicate occurrences of the same element in a multi-value.
- `@STRAT=options` (Optional) Strategy options governing the assignment (see Execute Library Overview for details).
- `@COMP=value-type` (Optional) Specifies the way in which the individual values in the multivalues are to be compared. (See Value Types below.)

At most one of the parameters @DUPLICATE and @NODUPLICATE may be specified; if neither is present, @NODUPLICATE is assumed by default.

Value Types

The comp specifier can be used for indicating how the individual values in a multivalue are to be compared. If it is absent, STRING is the default. The following types are valid: STRING, INT, FLOAT, DATE, and TIME.

For example, if one multivalue contains the element 1.0 and another multivalue contains the element 1.00, these will be regarded as the same value if @COMP=FLOAT is specified. However, if @COMP=STRING is specified (the default), they are regarded as two different strings.

Effect

If an @ADD parameter is given, each individual element in the @ADD list is added to the multi-value(s) designated by @ATOMID. If @DUPLICATE is specified, elements already present in the multi-value will be included again; if @NODUPLICATE, such additional occurrences will be suppressed.

If a @DELETE parameter is given, each individual element in the @DELETE list is deleted from the multi-value(s) designated by @ATOMID. If @DUPLICATE is specified, only the first occurrence of each element will be deleted, leaving any additional occurrences intact; if @NODUPLICATE, all occurrences of each element will be deleted.

Both @ADD and @DELETE may be specified in a single SetMultiValue. In that case, the deletes are done first.

Result

The result returned by SetMultiValue is TRUE if the call is successful, FALSE if an error occurs.

Examples

If Duckburg.residents contains the multi-value

Donald,Daisy,Dewey, a condition or action of the form

```
Execute "SetMultiValue" @ATOMID=Duckburg.residents;
                        @STRING="@ADD=Huey,Dewey,Louie";
```

will assign the string Donald,Daisy,Dewey,Huey,Louie as the new value of Duckburg.residents (since in the absence of any explicit indication, the default behavior is @NODUPLICATE). By contrast,

```
Execute "SetMultiValue" @ATOMID=Duckburg.residents;
                        @STRING="@ADD=Huey,Dewey,Louie,
                        @DUPLICATE";
```

will set it to Donald,Daisy,Dewey,Huey,Dewey,Louie, with the element Dewey duplicated. Following this operation,

```
Execute "SetMultiValue" @ATOMID=Duckburg.residents;
                        @STRING="@DELETE=Dewey";
```

will set Duckburg.residents to Donald,Daisy,Huey,Louie (defaulting to @NODUPLICATE and deleting all occurrences of the element Dewey), whereas

```
Execute "SetMultiValue" @ATOMID=Duckburg.residents;
```

```
@STRING="@DELETE=Dewey,@DUPLICATE";
```

will set it to Donald,Daisy,Huey,Dewey,Louie (deleting just the first occurrence of Dewey).

Related Topics

Multi-Values

Patterns

Execute Operator

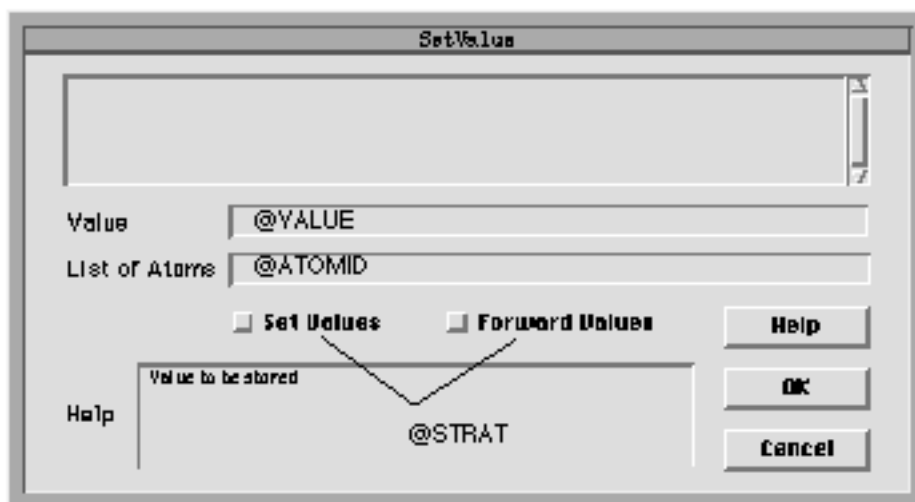
SetValue Routine

Definition

The Execute routine `SetValue` stores a fixed value into one or more designated slots (object properties).

Interactive Dialog

`SetValue` is chosen with the Select Execute popup menu command in the Rule editor or Method editor, which automatically displays a special dialog box for specifying the execute parameters interactively, rather than by explicitly typing them in as listed below:



Parameters

The `@ATOMID` parameter is a pattern specifying a list of objects or slots whose values are to be set.

The `@STRING` parameter may include the following:

`@VALUE=new_value` Value to be stored.

`@STRAT=options` (Optional) Strategy options governing the assignment (see Execute Library Overview for details).

The `@VALUE` parameter may specify a value of any type.

Effect

If @ATOMID represents a list of object properties, then all of the designated properties will be set to the value specified by @VALUE. If @ATOMID represents a list of objects themselves, then all properties of each such object will be set to the given value. Notice that this routine does not set the values associated directly with the objects themselves; if this is what is needed, the objects' Value property must be specified explicitly.

If the type of a property doesn't match that of the value to which it is to be set, the value is automatically converted to the required type. Some such conversions may not work properly, however (such as Date to Boolean); it is the application developer's responsibility to ensure that the specified assignments are meaningful.

Result

The result returned by SetValue is TRUE if the call is successful, FALSE if an error occurs.

Examples

Suppose class Box has two instances named box1 and box2 and two properties named width and height. Then a condition or action of the form

```
Execute "SetValue" @ATOMID=<Box>.width; @STRING="@VALUE=10";
```

will assign the value 10 to the properties box1.width and box2.width,

```
Execute "SetValue" @ATOMID=<Box>; @STRING="@VALUE=10";
```

will assign it to box1.width, box1.height, box2.width, and box2.height, and

```
Execute "SetValue" @ATOMID=<Box>.Value; @STRING="@VALUE=10";
```

will assign it directly to the objects box1 and box2 (that is, to the properties box1.Value and box2.Value).

Related Topics

[Execute Operator](#)

[Data Types](#)

[Value Property](#)

[Patterns](#)

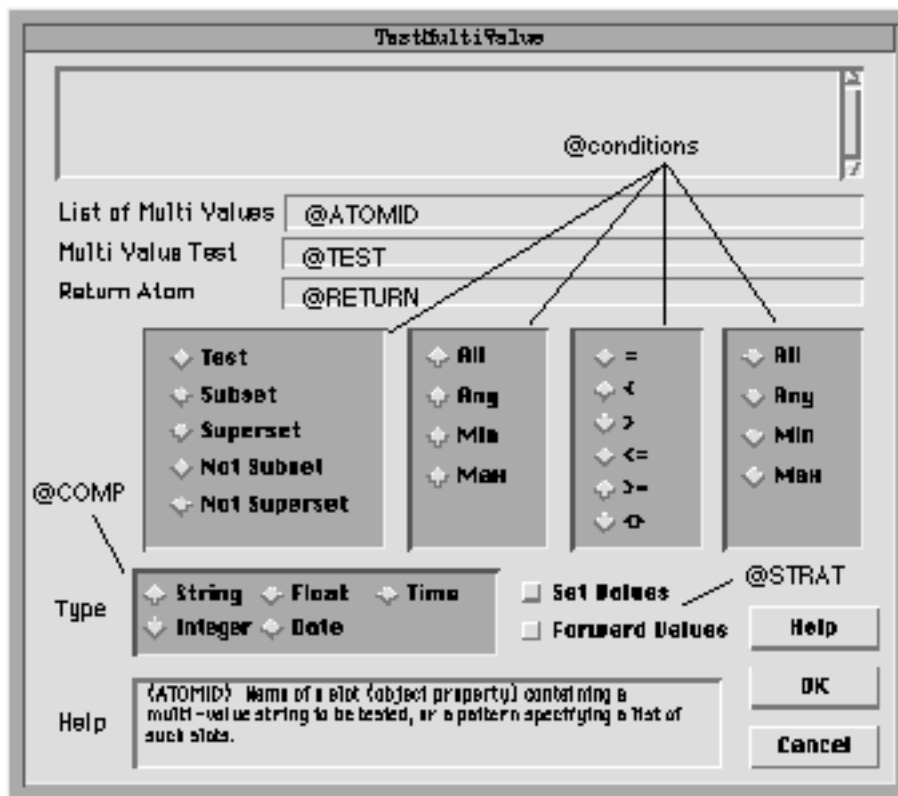
TestMultiValue Routine

Definition

The Execute routine TestMultiValue compares multi-values for a variety of possible relations.

Interactive Dialog

`TestMultiValue` is chosen with the Select Execute popup menu command in the Rule editor or Method editor, which automatically displays a special dialog box for specifying the execute parameters interactively, rather than by explicitly typing them in as listed below:



Parameters

The `@ATOMID` parameter is the name of a slot (object property) containing a multi-value string to be tested, or a pattern specifying a list of such slots.

The `@STRING` parameter may include the following:

<code>@TEST=</code> <i>test_val</i>	Slot containing multi-value to compare against.
<code>@condition</code>	Test to be applied (see Test conditions below).
<code>@RETURN=</code> <i>answer</i>	Destination in which to return result of test.
<code>@STRAT=</code> <i>options</i>	(Optional) Strategy options governing the assignment (see Execute Library Overview for details).
<code>@COMP=</code> <i>value-type</i>	(Optional) Specifies the way in which the individual values in the multivalues are to be compared. (See Value Types below.)

The destination specified by @RETURN must be either a boolean-valued slot, the name of a class, or the name of an object. If it is a boolean slot, then @ATOMID must also designate a single slot (rather than a pattern matching a whole list of slots).

Value Types

The comp specifier can be used for indicating how the individual values in a multivalue are to be compared. If it is absent, STRING is the default. The following types are valid: STRING, INT, FLOAT, DATE, and TIME.

For example, if one multivalue contains the element 1.0 and another multivalue contains the element 1.00, these will be regarded as the same value if @COMP=FLOAT is specified. However, if @COMP=STRING is specified (the default), they are regarded as two different strings.

Test conditions

The test condition included in the @STRING parameter specifies the type of comparison to be performed on the multi-values. It consists of one of the four keywords

MIN	Smallest element
MAX	Largest element
ANY	Any element
ALL	All elements

followed by one of the six comparison operators

=	Equal
<>	Not equal
<	Less than
<=	Less than or equal
>	Greater than
>=	Greater than or equal

followed by another of the four keywords. The first keyword refers to the multi-value specified by the @ATOMID parameter, the second to that specified by the @TEST parameter. Thus, for example, the test condition @MIN>MAX tests whether the smallest element of @ATOMID is greater than the largest element of @TEST. Thus 96 test conditions are possible (though some of them turn out to have the same meaning):

@MIN=MIN	@MAX=MIN	@ANY=MIN	@ALL=MIN
@MIN=MAX	@MAX=MAX	@ANY=MAX	@ALL=MAX
@MIN=ANY	@MAX=ANY	@ANY=ANY	@ALL=ANY
@MIN=ALL	@MAX=ALL	@ANY=ALL	@ALL=ALL
@MIN<>MIN	@MAX<>MIN	@ANY<>MIN	@ALL<>MIN
@MIN<>MAX	@MAX<>MAX	@ANY<>MAX	@ALL<>MAX
@MIN<>ANY	@MAX<>ANY	@ANY<>ANY	@ALL<>ANY
@MIN<>ALL	@MAX<>ALL	@ANY<>ALL	@ALL<>ALL

@MIN<MIN	@MAX<MIN	@ANY<MIN	@ALL<MIN
@MIN<MAX	@MAX<MAX	@ANY<MAX	@ALL<MAX
@MIN<ANY	@MAX<ANY	@ANY<ANY	@ALL<ANY
@MIN<ALL	@MAX<ALL	@ANY<ALL	@ALL<ALL
@MIN<=MIN	@MAX<=MIN	@ANY<=MIN	@ALL<=MIN
@MIN<=MAX	@MAX<=MAX	@ANY<=MAX	@ALL<=MAX
@MIN<=ANY	@MAX<=ANY	@ANY<=ANY	@ALL<=ANY
@MIN<=ALL	@MAX<=ALL	@ANY<=ALL	@ALL<=ALL
@MIN>MIN	@MAX>MIN	@ANY>MIN	@ALL>MIN
@MIN>MAX	@MAX>MAX	@ANY>MAX	@ALL>MAX
@MIN>ANY	@MAX>ANY	@ANY>ANY	@ALL>ANY
@MIN>ALL	@MAX>ALL	@ANY>ALL	@ALL>ALL
@MIN>=MIN	@MAX>=MIN	@ANY>=MIN	@ALL>=MIN
@MIN>=MAX	@MAX>=MAX	@ANY>=MAX	@ALL>=MAX
@MIN>=ANY	@MAX>=ANY	@ANY>=ANY	@ALL>=ANY
@MIN>=ALL	@MAX>=ALL	@ANY>=ALL	@ALL>=ALL

In addition, four special test conditions are recognized

@SUBSET @SUPERSET @NOT_SUBSET @NOT_SUPERSET

making a total of 100 possible test conditions in all.

The following chart shows how many of the executes have related meanings. The operators in the left column replace the asterisk (*) in the expressions along the top row. All of the operations in a given box have the same meaning. For example, the following three operations have the same effect when used in TestMultiValue: MAX>MIN, ANY>MIN, MAX>ANY.

In addition, these are the same as ANY>ANY, which is shown in the column header.

M * T	ANY * ANY	ANY * ALL	ALL * ANY	ALL * ALL
=	$\exists m \in M / \exists t \in T, m=t$	$\forall t \in T / \exists m \in M, t=m$ SUPERSET	$\forall m \in M / \exists t \in T, m=t$ SUBSET	$\forall m \in M / \exists t \in T, m=t$ AND $\forall t \in T / \exists m \in M, t=m$
<>	$\exists m \in M / \exists t \in T, m \neq t$	$\exists m \in M / \forall t \in T / m \neq t$ NOT_SUBSET	$\exists t \in T / \forall m \in M, t \neq m$ NOT_SUPERSET	$\forall m \in M / \forall t \in T, m \neq t$
>	MAX>MIN ANY>MIN MAX>ANY	MAX>MAX ANY>MAX MAX>ALL	MIN>MIN ALL>MIN MIN>ANY	MIN>MAX ALL>MAX MIN>ALL
<	MIN<MAX ANY<MAX MIN<ANY	MIN<MIN ANY<MIN MIN<ALL	MAX<MAX ALL<MAX MAX<ANY	MAX<MIN ALL<MIN MAX<ALL
>=	MAX>=MIN ANY>=MIN MAX>=ANY	MAX>=MAX ANY>=MAX MAX>=ALL	MIN>=MIN ALL>=MIN MIN>=ANY	MIN>=MAX ALL>=MAX MIN>=ALL
<=	MIN<=MAX ANY<=MAX MIN<=ANY	MIN<=MIN ANY<=MIN MIN<=ALL	MAX<=MAX ALL<=MAX MAX<=ANY	MAX<=MIN ALL<=MIN MAX<=ALL

Effect

As noted above, if the @RETURN parameter designates a boolean slot, then @ATOMID must also be a single slot containing a multi-value string. The multi-values specified by @ATOMID and @TEST are compared according to the given test condition, and the boolean result is stored into the slot specified by @RETURN.

If the @RETURN parameter instead designates a class or an object, then @ATOMID may be either a single slot containing a multi-value string or a pattern matching a whole list of such slots. Each multi-value in turn is compared with the one specified by the @TEST parameter, using the given test condition. If @RETURN is a class, all multi-values for which the result of

the test is `TRUE` are added to it as instances; if it is an object, they are associated with it as components (subobjects).

Result

The result returned by `TestMultiValue` is `TRUE` if the call is successful, `FALSE` if an error occurs.

Examples

In all of the following examples, `TheAnswer` is a boolean-valued object and `ABC`, `BCD`, and `CDE` are instances of class `Alphabet` with the following initial values:

```
ABC.members = "alpha,beta,charlie"
BCD.members = "beta,charlie,dog"
CDE.members = "charlie,dog,echo"
```

Example 1

A condition or action of the form

```
Execute "TestMultiValue" @ATOMID=ABC.members ;
                        @STRING="@TEST=alpha,charlie,beta,
                        @ALL=ALL,@RETURN=TheAnswer.Value" ;
```

will set `TheAnswer.Value` to `TRUE`, since all elements in `ABC.members` equal all elements in `@TEST`. (Notice that the order in which the elements are given is unimportant.) However,

```
Execute "TestMultiValue" @ATOMID=ABC.members ;
                        @STRING="@TEST=alpha,beta,
                        @ALL=ALL,@RETURN=TheAnswer.Value" ;
```

sets `TheAnswer.Value` to `FALSE` (since `ABC.members` contains elements that are not matched by those in `@TEST`), and

```
Execute "TestMultiValue" @ATOMID=ABC.members ;
                        @STRING="@TEST=alpha,beta,gamma,
                        @ALL=ALL,@RETURN=TheAnswer.Value" ;
```

also sets it to `FALSE` (since `@TEST` contains elements that don't match those in `ABC.members`).

Example 2

```
Execute "TestMultiValue" @ATOMID=ABC.members ;
                        @STRING="@TEST=alpha,gamma,
                        @ANY=ANY,@RETURN=TheAnswer.Value" ;
```

sets `TheAnswer.Value` to `TRUE` (since `ABC.members` contains at least one element that matches at least one element in `@TEST`), but

```
Execute "TestMultiValue" @ATOMID=ABC.members ;
                        @STRING="@TEST=gamma,delta,
                        @ANY=ANY,@RETURN=TheAnswer.Value" ;
```

sets it to `FALSE` (since `ABC.members` and `@TEST` have no elements in common).

Example 3

```
Execute "TestMultiValue" @ATOMID=ABC.members ;
                        @STRING="@TEST=alpha,delta,
                        @ANY>ANY,@RETURN=TheAnswer.Value" ;
```

sets `TheAnswer.Value` to `TRUE`, since `ABC.members` contains at least one element (beta) that is greater than at least one element in `@TEST` (alpha), but

```
Execute "TestMultiValue" @ATOMID=ABC.members ;
                        @STRING="@TEST=alpha,delta,
                        @ALL>ALL,@RETURN=TheAnswer.Value" ;
```

sets it to `FALSE`, since not all elements in `ABC.members` are greater than all elements in `@TEST`.

Example 4

```
Execute "TestMultiValue" @ATOMID=ABC.members ;
                        @STRING="@TEST=alpha,beta,gamma,
                        @MAX>MIN,@RETURN=TheAnswer.Value" ;
```

sets `TheAnswer.Value` to `TRUE`, since the largest element in `ABC.members` (charlie) is greater than the smallest element in `@TEST` (alpha), but

```
Execute "TestMultiValue" @ATOMID=ABC.members ;
                        @STRING="@TEST=alpha,beta,gamma,
                        @MIN>MAX,@RETURN=TheAnswer.Value" ;
```

sets it to `FALSE`, since the smallest element in `ABC.members` (alpha) is not greater than the largest element in `@TEST` (gamma), and

```
Execute "TestMultiValue" @ATOMID=ABC.members ;
                        @STRING="@TEST=alpha,beta,gamma,
                        @MAX<MIN,@RETURN=TheAnswer.Value" ;
```

also sets it to `FALSE`, since the largest element in `ABC.members` (charlie) is not less than the smallest element in `@TEST` (alpha).

Example 5

```
Execute "TestMultiValue" @ATOMID=ABC.members ;
                        @STRING="@TEST=alpha,beta,
@SUPERSET,@RETURN=TheAnswer.Value" ;
```

sets `TheAnswer.Value` to `TRUE` (since `ABC.members` is a superset of `@TEST`), and

```
Execute "TestMultiValue" @ATOMID=ABC.members ;
                        @STRING="@TEST=alpha,beta,gamma,charlie,
                        @SUBSET,@RETURN=TheAnswer.Value" ;
```

also sets it to `TRUE` (since `ABC.members` is a subset of `@TEST`), but

```
Execute "TestMultiValue" @ATOMID=ABC.members ;
                        @STRING="@TEST=alpha,gamma,
                        @SUBSET,@RETURN=TheAnswer.Value" ;
```

sets it to `FALSE` (since in this case `ABC.members` is not a subset of `@TEST`).

Example 6

```
Execute "TestMultiValue" @ATOMID=<Alphabet>.members ;
                        @STRING="@TEST=apple,candy,
                        @ANY<ANY,@RETURN=TheAnswer" ;
```

associates the objects `ABC` and `BCD` as components (subjects) of `TheAnswer`, since they each contain at least one element that is less than at least one element of the `@TEST` multi-value. However, `CDE.members` contains no such element, so `CDE` is not made a component of `TheAnswer`.

```
Execute "TestMultiValue" @ATOMID=<Alphabet>.members;
                               @STRING="@TEST=dog,SUPERSET,
                               @RETURN=TheAnswer";
```

makes BCD and CDE components of TheAnswer, since they are both supersets of @TEST, but ABC is not.

```
Execute "TestMultiValue" @ATOMID=<Alphabet>.members;
                               @STRING="@TEST=alpha,echo,
                               @ANY=ANY,@RETURN=TheAnswer";
```

makes ABC and CDE components of TheAnswer, since they each contain at least one element (alpha and echo, respectively) that is equal to some element of @TEST. However, BCD.members contains no such element, so BCD is not made a component of TheAnswer.

Related Topics

Comparison Operators

Multi-Values

Patterns

Execute Operator

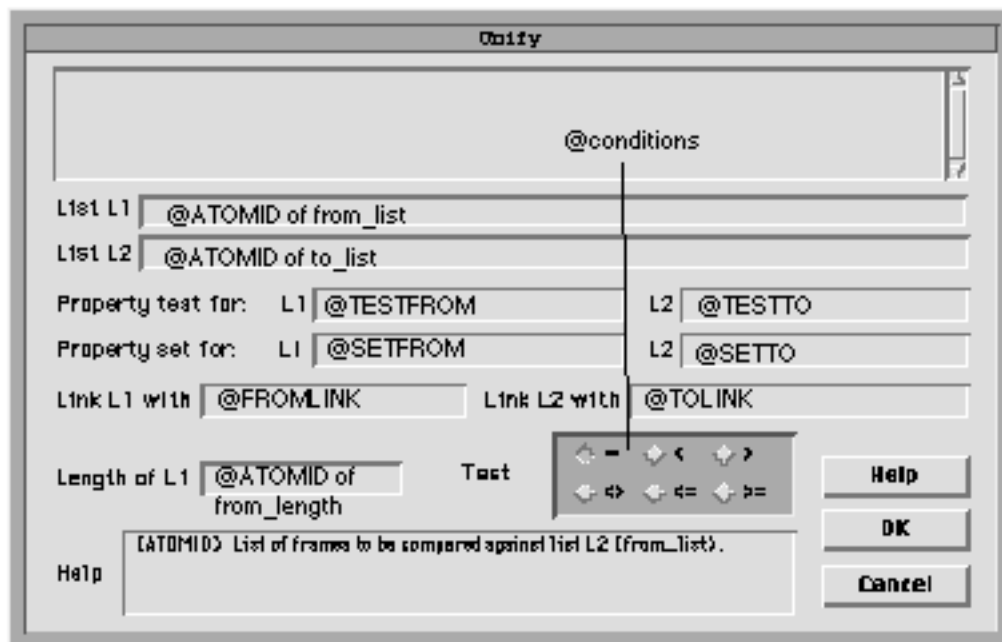
Unify Routine

Definition

The Execute routine `Unify` compares specified properties of two lists of frames (objects or classes) and finds those pairs that satisfy a stated condition.

Interactive Dialog

`Unify` is chosen with the Select Execute popup menu command in the Rule editor or Method editor, which automatically displays a special dialog box for specifying the execute parameters interactively, rather than by explicitly typing them in as listed below:



Parameters

The @ATOMID parameter consists of the following items, separated by commas:

<i>from_length</i>	Number of frames in <i>from_list</i> .
<i>from_list</i>	List of frames to be compared.
<i>to_list</i>	List of frames to compare to.

The @STRING parameter may include the following:

@TESTFROM= <i>from_prop</i>	Relevant property of <i>from_list</i> .
@TESTTO= <i>to_prop</i>	Relevant property of <i>to_list</i> .
@ <i>condition</i>	Test condition to be applied (see Test conditions below).
@SETFROM= <i>set_from_prop</i>	(Optional) Property to copy from <i>from_list</i> .
@SETTO= <i>set_to_prop</i>	(Optional) Property to copy to in <i>to_list</i> .
@FROMLINK= <i>from_link_frame</i>	(Optional) Frame in which to accumulate <i>from_list</i> elements.
@TOLINK= <i>to_link_frame</i>	(Optional) Frame in which to accumulate <i>to_list</i> elements.

The @TESTFROM and @TESTTO parameters are required. All others are optional, but the pairs @SETFROM/@SETTO and @FROMLINK/@TOLINK must be specified together: that is, if one of the pair is present, the other must be present as well.

Test Conditions

The test condition included in the @STRING parameter specifies the type of comparison to be performed. It must consist of exactly one of the following:

EQUAL	Equality
NOT_EQUAL	Inequality
LESS	Less-than
LESS_EQUAL	Less-than-or-equal
GREATER	Greater-than
GREATER_EQUAL	Greater-than-or-equal

If no test condition is specified, @EQUAL is assumed by default.

Effect

The value of property *from_prop* for each frame in *from_list* is compared with that of property *to_prop* for each frame in *to_list*, using

the stated test condition. If the condition holds and the parameters @SETFROM and @SETTO are specified, then the value of property `set_from_prop` in the `from_list` element is copied to property `set_to_prop` in the `to_list` element. In addition, if @FROMLINK and @TOLINK are specified, then the `from_list` element is attached to `from_link_frame` as an instance or component, and the `to_list` element is similarly attached to `to_link_frame`.

This behavior is summarized by the following fragment of pseudo-code:

```
for each from_frame in from_list
  for each to_frame in to_list
    if from_frame.from_prop <condition>
to_frame.to_prop
      assign from_frame.set_from_prop to
to_frame.set_to_prop
      attach from_frame to from_link_frame
      attach to_frame to to_link_frame
    end if
  end for
end for
```

Result

The result returned by Unify is TRUE if the call is successful, FALSE if an error occurs.

Examples

Example1: Suppose we have a class `Pianos` with property `width` and a class `Doorways` with property `height`. Since a piano must be tilted on its side to get through a door, the width of the piano must be less than the height of the door. A condition or action of the form

```
Execute "Unify"
@ATOMID=numPianos.Value, <Pianos>, <Doorways>;
    @STRING="@LESS, @TESTFROM=width,
    @TESTTO=height, @SETFROM=model,
    @SETTO=accommodates,
    @FROMLINK=Small_enough_pianos,
    @TOLINK=Big_enough_doors";
```

will test the width of each piano against the height of each door to see if it will fit. If, say, `Grand_Piano.width` is less than `Front_Door.height`, then `Grand_Piano` will become an instance of class `Small_enough_pianos`, `Front_Door` will become an instance of `Big_enough_doors`, and the value of `Grand_Piano.model` (Steinway, for example) will be assigned to `Front_Door.accommodates`.

Example 2: Suppose we have a class `Truck_Drivers` with properties `city` and `name`, and a class `Trucks` with properties `location` and `driver`. In order for a truck to be driven, there must be an available driver in the same city. A condition or action of the form:

```
Execute "Unify"    @ATOMID=numTruckDrivers.Value, <Truck_Drivers>,
<Trucks>;
    @STRING="@EQUAL, @TESTFROM=city, @TESTTO=location,
    @SETFROM=name, @SETTO=driver, @FROMLINK=Can_Drive,
    @TOLINK=can_go";
```

will test the city of each `Truck_Driver` against the location of each `Truck` to see if they match. If, for example, `Chuck.city` and `Ace.location` are both "Chicago", then `Chuck` will become an instance of class `Can_Drive`,

Ace will become an instance of class `Can_Go`, and the value of `Chuck.name` (Charles Smith, for example) will be assigned to `Ace.Driver`.

It is important to note that:

- The length of the first list (`from_length`) can be obtained in a condition directly before the `Unify` execute by using the `Length` function.
- When a match is found in a `Unify`, the appropriate assignments take place and no further matches are sought on that object! For example, once we have found a driver for a truck, no further searching is done on that truck, even if several drivers are available in the same city.

Related Topics

Patterns

Execute Operator

Length Function

Comparison Operators

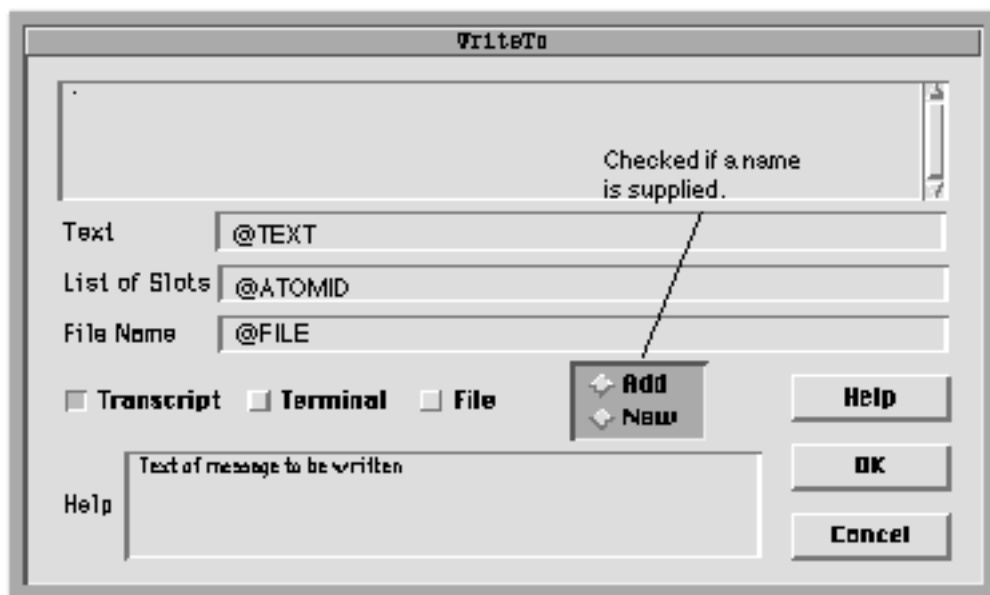
WriteTo Routine

Definition

The Execute routine `WriteTo` writes a message to the transcript, a file, or the terminal.

Interactive Dialog

`WriteTo` is chosen with the Select Execute popup menu command in the Rule editor or Method editor, which automatically displays a special dialog box for specifying the execute parameters interactively, rather than by explicitly typing them in as listed below:



Parameters

The @ATOMID parameter is an optional list of slots (object properties) whose values are to be appended to the message.

The @STRING parameter may include the following:

@TEXT= <i>text_string</i>	Text of message to be written.
@TRANSCRIPT	(Optional) If present, write message to transcript.
@FILE= <i>file_name</i>	(Optional) If present, write message to specified file.
@TERMINAL	(Optional) If present, write message to terminal.
@ADD	(Optional) If present, append message to existing file.
@NEW	(Optional) If present, create new file.

Any combination of the parameters @TRANSCRIPT, @FILE, and @TERMINAL may be included, but at least one must be present. If neither @ADD nor @NEW is specified, @ADD is assumed by default.

Effect

The message given by the @TEXT parameter is written to the transcript, a file, and/or the terminal, as specified by the parameters. If a list of slots is specified with @ATOMID, their names and current values are written after the end of the message text.

The @ADD and @NEW options are meaningful only if a file name is given with @FILE. @ADD appends the message to the end of the designated file; if the file does not exist, it is created automatically. @NEW forces creation of a new file containing the specified message; if an old file already exists with the same name, it is converted to a \$\$\$backup file.

Result

The result returned by WriteTo is TRUE if the call is successful, FALSE if an error occurs.

Examples

A condition or action of the form

```
Execute "WriteTo" @STRING="@TEXT=Failure in Valve #3,
@TRANSCRIPT";
```

will write the message Failure in Valve #3 to the transcript.

A condition or action of the form

```
Execute "WriteTo" @STRING="@TEXT=Tank pressures are ,
@TRANSCRIPT,@FILE=Session.log";
@ATOMID=<Tank>.pressure;
```

will append the message Tank pressures are to both the transcript and the file Session.log, followed by the values of the property pressure for all instances of class Tank.

Related Topics

Patterns

Execute Operator

Database Integration Topics

This chapter describes the various procedures, key concepts, and general principles of the Rules Element database interface. The topics appear in alphabetical order.

Core Database Topics

New users should read these first for more detailed information about the different ways the database interface can be used and for specific information about specific database types.

Database Interface Features

Identifies features of the Rules Element database interface that you can use to extend the database retrieve and write capabilities of your knowledge-based application.

Rule Editor / Method Editor Windows

Lists topics related to setting up database retrieve / write operations in a rule or method.

Database Editor Windows

Lets you find descriptions of the Database Editor windows' various fields.

Database Interface Operations

The topics in this list identify optional as well as required tasks of the retrieve / write operations. This information supplements the Database Editor Windows topics list.

Before looking up topics in this chapter read Chapter Seven, "Application Data" in the Intelligent Rules Element User's Guide.

Access String

General

When the Rules Element begins a retrieve or write operation, it first needs to access the file or database server containing the data to be accessed.

The first argument of a Retrieve or Write command is a quoted string which specifies the database access string used to establish communications with the database. This string can be as simple as just a filename or something more complex, for example containing one or more of the following fields: username, password, server name, database name, network transport mechanism, or computer node name. Typically, the more complicated access strings are used by relational databases.

Related Topics

- Access String Specification
- Retrieve Operator
- Write Operator

Access String Specification

General

To supply the database access string field enter the name of the file (for flat file databases) or the database access string (for relational databases) as the first argument of the Retrieve or Write operator. A quoted entry for this field is required for the Rules Element to initiate the desired operation. To pass a null string specify "" (double quotes) as the first argument of the Retrieve or Write operator.

The following example shows how the database access string would be specified for a Retrieve operation. In this example, the access string "scott tiger" appears as the Retrieve operator's first argument in the Rule Editor window.



Figure 3-1 Specifying a Database Access String

Flat-File Databases

For flat-file databases, the string is interpreted as a file name, and is handled like any other file name on your operating system. For flat-file databases such as NXPDB and DBASE III files, the access string must contain the

filename of the data file or database. The filename extension is optional. If it is not specified, the Rules Element uses the following default extensions:

- .NXP and NXPDB files
- .SLKSYLK (Excel) files
- .DBFDBase III files
- .WKS files (Lotus 1-2-3)

Relational Databases

For relational databases such as Oracle, Sybase, and Ingres, the Rules Element needs the access string used to sign on to the database manager. The string you supply is passed to the database manager for interpretation. Parameters in the connection string must be delimited by a space character.

You must not skip parameters within the access string, but you can omit the last parameter in the string. If you need to, use a dummy name to supply a connection parameter that is not used, but do not skip a parameter or replace one by blanks. For example, in the case of Sybase, the connection string might take the following form:

```
"scott tiger hyperion SYBASE_HYPERION MyApp customerdb"
```

In this example, the application name `MyApp` was supplied as a dummy placeholder.

Details about specific database access string requirements are located in the corresponding database name topic in this manual.

Pathname Specification

Absolute or relative pathnames can be used. The pathname syntax depends on the underlying operating system:

```
For DOS: \dir1\dir2\file1
For UNIX: /dir1/dir2/file1
For VMS: $disk:[dir1.dir2]file1
```

If a relative filename specification is used, the Rules Element will use its own search path (the logical name `ND_PATH` on VMS, the shell variable `ND_PATH` on UNIX, or the path specified in the Rules Element on the Macintosh) to locate flat-file databases. The filename will be concatenated to each of the directories in the search path until a file is found.

Environment Variables

On some systems (VMS & UNIX), the Rules Element will attempt to construct the access string argument using environment variables. On VMS systems, you can specify a VMS logical name. On UNIX systems, you can use shell environment variables (setenv(1) or EXPORT variables). This feature is particularly useful with ORACLE, as the account/password information can be hidden in an environment variable.

Dynamic Values

It's possible to use Rules Element interpretations ("`@V(...)`") in this field. Instead of having a fixed value, the string can be constructed at runtime

from the string values of various object slots. The quoted string can contain any of the following constructs:

@V(obj.prop)	will be replaced by the current value of obj.prop
@SELF	will be replaced by the name of the current object (methods only)
@PROP	will be replaced by the name of the current property (methods only)

For example, when working with flat-file databases different cases can be stored in files called `filecase1`, `filecase2`, etc. If you specify "`@V(cur_case.filename)`", and `cur_case.filename` currently holds the value `filecase2`, then the file `filecase2` will be used for the transaction.

Note: Slot names used in `@V(obj.prop)` constructs are not compiled when the rule or method containing the retrieve or write statement is compiled, they are interpreted at runtime. Usually, these slots exist elsewhere in the knowledge base, but if you misspell a name in these special constructs, the Rules Element will not create the corresponding object or slot and you will get an error at runtime.

Related Topics

Write Operator	Oracle
Retrieve Operator	Sybase
Interpretations @V(...)	Informix
Dynamic Values	Ingres

Also, look up your database type for details about the exact syntax required.

Arguments Overview

The Retrieve and Write operators invoke a Database Editor window that provides fields to specify the retrieve or write operation. The following two lists give an overview of the fields for the Retrieve window and the Write window.

Retrieve Arguments

The following table summarizes the various arguments available in a Retrieve operation:

Database Type	Always required. Indicates type of database to retrieve from. The default type is machine dependent.
Name	Name of the object to be updated or created when reading the current record. If the object already exists, the record is retrieved (see the In filter) into the object. If the object does not exist and CreateObject is checked, a dynamic object is created. If the object does not exist and CreateObject is not checked, the record is skipped. This field is not normally used for sequential queries or atomic queries.

In	List of objects, classes, slots used to filter what is to be retrieved. If empty list (the default), all fields are retrieved. If non-empty list, only those fields mapped to objects or slots in the lists are retrieved. Pattern matching lists or interpretations can be used. Used in grouped or atomic transactions.
Link To	List of classes or objects to which the new or updated object should be linked. Pattern matching lists or interpretations can be used.
Cursor	For sequential retrieves, an atom of type integer that represents the record number or the query number. For atomic retrieves, it must be present, but specified as UNKNOWN. For grouped retrieves, it must be empty.
Begin	Box for the command starting the transaction of a relational database. Executed only once for sequential retrieves. Also used to hold the range name for the SYLKDB (Excel) and WKSDB (Lotus123) types.
End	Box for end of transaction command (typically used for Write). End statement is only done the last time during a sequential operation.
Create Object	Enables the creation of dynamic objects when the current Name doesn't exist in the KB. Valid for grouped retrieve operations only.
Fields / Props List	Describes how to map the fields of each record to the property-slots of the object in the Rules Element. If the lists are empty, ALL property names will be used as Field names. If the lists are not empty, only the Fields / Properties specified are affected. For atomic or sequential retrieves, the Rules Element list should be slots (obj.prop). Otherwise, a list of properties belonging to the object being retrieved into should be provided.
Query	Box for the actual database query. Look up your database type for details. Query is used to select the current record in sequential mode. The cursor refers to a query id in that case.
Retrieve Unknown	Check this option to enable UNKNOWN values to be read (string "Unknown" in the database or spreadsheet). The default is to have the option off so that only meaningful values are retrieved.
Retrieve Strategies	Specify the way values are propagated after a Retrieve. Always Forward means values are used in the forward chaining. Current Forward means the current strategy of the rule is used (this is the default). Do Not Forward means the values are used without effects. Note that the If Change methods are also triggered when new values are retrieved.

Write Arguments

The following table summarizes the various arguments available in a Write operation:

Database Type	Always required. Indicates type of database to write to. The default type is machine dependent.
Name	Name of the object to use for updating the current record. If the object already exists, the record is updated (see the In filter), otherwise the record is skipped. This field is not normally used for sequential queries or atomic queries.
In	Filters records to be written back. If empty list (the default), all fields are updated. If non-empty list, only those fields mapped to objects or slots in the lists are written. All remaining atoms in the list create new records (if Create Record selected). Pattern matching lists or interpretations can be used. Usually not used in sequential transactions.
Cursor	An atom of type integer that represents the record number or the query number. Typically used for sequential write(s) following sequential retrieves.
SqlError	Name of the slot in the knowledge base that you want to use to trap Sql database server error messages.
Fields / Props List	Describes how to map the fields of each record to the property-slots of the object in the Rules Element. If the lists are empty, ALL property names will be used as Field names. If the lists are not empty, only the Fields / Properties specified are affected. For atomic or sequential writes, the Rules Element list should be slots (obj.prop). Otherwise, a list of properties belonging to the object being written should be provided.
Create New Record	Enables the creation of new records with the atoms in the In list not already used, otherwise updates the records that already exist. Valid for grouped write operations only.
Insert Only	Assumes no records exist in the database to correspond to the atoms in the In list and enables the creation of all new records. Not used for sequential transactions.
New File	Instead of updating an existing file, a new file is created with a set of new records. Note that if the In list is empty, each object of the KB is written in a record. And if the Fields and Properties lists are empty, all existing properties are used. Cannot be used in sequential write or with relational databases.

Begin	Box for the command starting the transaction of a relational database. Executed only once for sequential transactions. Also used to hold the range name for the SYLKDB (Excel) and WKSDB (Lotus123) types.
Query	Box for the actual database query. Look up your database type for details. Query is used to select the current record in sequential mode. The cursor refers to a query id in that case.
End	Box for end of transaction command (typically used for Write). End statement is only done the last time during a sequential operation.
Write Unknown	Check this option to enable UNKNOWN values to be written (string "Unknown" in the database or spreadsheet). The default is to have the option off so that only meaningful values are stored.

Related Topics

Database Editor Windows
Retrieve Operator
Write Operator

Also, look up individual arguments and your database type for more detailed information.

Atomic Retrieve

General

Atomic retrieval can be used with both flat-file databases and relational databases such as INGRES, Sybase, and Oracle.

An atomic retrieve operation reads the fields from one record (and only one record) into slots in the Rules Element's working memory. The slots (object.property combinations) usually all belong to the same object, but it's also possible to read the fields into slots belonging to two or more objects.

Atomic retrieves are used when the knowledge base needs to retrieve a single, isolated bit of information about the problem at hand. For example, an atomic read would be used to get a single car's *Price* and *Sportive* fields from the *CARS* database.

Atomic retrieves are also "isolated" from the standpoint that they don't need any "surrounding" logic in the knowledge base or object network to support them. A retrieve can be included in the LHS or RHS of any rule without regard for it affecting other rules in the knowledge base. Of course, if the retrieve is used in the LHS of a rule and it returns "False", then execution of the LHS will be terminated and the rule's hypothesis will be set to "False".

Specification

The Rules Element recognizes atomic retrieves from the fact that a cursor slot is provided in the database retrieve window, and it has the value UNKNOWN when the retrieve is executed. If the cursor's value is NOT

unknown, the Rules Element will assume that the retrieve is sequential and unpredictable results will occur.

To determine which record will be retrieved, a query is included in the database retrieve window's Query field. The query should be specific enough to return one, and only one record to the Rules Element. If the query isn't specific enough and more than one record is returned, only the first record is processed. For relational databases, you can use any query accepted by the database manager (usually an ANSI SQL statement), for flat-file databases, you can use the Rules Element Query Language to filter the records.

If the query fails and no record is returned by the retrieve, the cursor slot is set to -1. If the query succeeds and the record is retrieved, the cursor is set to an arbitrary positive number.

Atomic retrieves always read the record's fields into specific slots which already exist when the retrieve is issued. As a rule, objects are not created by atomic retrieves.

Fields

To build an atomic retrieve, complete the Retrieve screen in the Database Editor window as follows.

- Ensure that the cursor slot which will be specified in the retrieve window has an "Unknown" value. An easy way to do this is to include a "Reset slot_name" (where "slot_name" is the cursor's slot name) before the retrieve operation.
- Specify Retrieve as the operator for the LHS, RHS, if change, or order of sources statement.
- As the first operand of the Retrieve, specify the database access string for the relational database being accessed.
- In the database retrieve window, click on the appropriate selection in the Database Type field for the database being retrieved from.
- The Begin field should contain whatever is appropriate for your database.
- In the Query field specify the database table name and appropriate SQL query OR the Rules Element query to select the record to be retrieved.
- The End field should contain whatever is appropriate for your database to end a transaction.
- The Name field should be left blank.
- The Cursor field should contain the name of the slot to be used as the cursor for this retrieve operation. This slot must be of the integer type, and MUST have an "Unknown" value when the retrieve is issued. The slot name may be specified as "object.property" or just "object", which is shorthand for "object.Value".
- The In field should contain the name of the slot that will update the database record field.
- The Link to field should be left empty
- In the Database Fields column, specify the names of the database fields to be retrieved. In the corresponding Object Properties column, specify the property slots into which the fields should be retrieved.

- The Create Object option must be left unselected. Only grouped retrieves can be used to create objects.

Related Topics

Cursor Slot Specification	Retrieving from Databases
Access String	Slot Specification for Retrieves
Query Retrieve Operations	Object Names In Retrieve Operations
Database Editor Windows	

Also, look up individual arguments and your database type for more detailed information.

Atomic Write

General

Atomic write operations can be used with both flat-file databases and relational databases such as INGRES, Sybase, and Oracle.

An atomic write takes the slots from one or more objects and writes them out to fields in a database record. In the vast majority of the cases, the slots are written to a single record, but it's also possible to update multiple records with an atomic write operation. The fields can all be written from slots which belong to the same object, or from slots belonging to several objects. When all the slots are written from the same object, the object-property relationship is, in effect, transformed into a record-field relationship.

Atomic writes are used to write out a single piece of information from the Rules Element's working memory. For example, an atomic write could be used in a slot's If Change actions to update a field in a database record when a slot's value changes.

For example, a knowledge base which recalculates the `Price` properties of cars (to apply a discount) could use an atomic write to update the `CARS` database with new `DB_PRICE` field values.

Atomic writes are "isolated" from the standpoint that they don't need any "surrounding" logic in the knowledge base or object network to support them. An atomic write can be included in the LHS or RHS of any rule without regard for it affecting other rules in the knowledge base. Of course, if the write is used in the LHS of a rule and it returns "False" (due to an error), then execution of the LHS will be terminated and the rule's hypothesis will be set to "False".

Specification

An atomic write is recognized by the fact that a cursor slot is provided in the database write window, and it has an 'UNKNOWN' value when the write is executed. If the cursor's value is NOT unknown, the Rules Element will assume that the write is sequential and unpredictable results will occur.

To determine which record's fields will receive the slots, a query is included in the database write window's Query field. It is very important that the query be specific enough to update ONLY the intended records. If the

query is not specific enough, then many more records could be updated than intended. For relational databases, you can use any query accepted by the database manager (usually an ANSI SQL statement), for flat-file databases, you can use the Rules Element Query Language to filter the records.

The Rules Element implements atomic writes by building a SQL UPDATE statement with a WHERE clause constructed from the Query field. For example, if the Query field contained:

```
CARS WHERE DB_CAR_NAME='car_1'
```

the SQL statement would look like:

```
UPDATE CARS SET . . . WHERE DB_CAR_NAME='car_1'
```

It's also possible to use a "parameterized query" which substitutes data from the knowledge base into the query at execution time.

If the query fails and no records are updated, the cursor slot is set to -1. If the query succeeds and record(s) are updated, the cursor is set to an arbitrary positive number.

Atomic writes always update existing records. Atomic writes cannot be used to add new records.

Fields

To build an atomic write, complete the Write screen in the Database Editor window as follows.

- Ensure that the cursor slot which will be specified in the Write window has an "Unknown" value. An easy way to do this is to include a "Reset slot_name" (where "slot_name" is the cursor's slot name) before the write operation.
- Specify Write as the operator in the LHS or RHS of the rule.
- As the first operand of the Write, specify the database access string for the relational database being accessed.
- In the database write window, click on the appropriate selection in the Database Type field for the database being retrieved from.
- The Begin field should contain whatever is appropriate for your database. See the Beginning Database Operations topic for more information.
- In the Query field specify the database table name and appropriate SQL query OR the Rules Element query to select the record to be updated.
- The End field should contain whatever is appropriate for your database to end a transaction. For almost all relational databases, either "COMMIT" or "COMMIT RELEASE" should be specified.
- The Name field may be left blank or may contain an explicit object name whose property slots will be written to the record's fields.
- The Cursor field should contain the name of the slot to be used as the cursor for this write operation. This slot must be of the integer type, and MUST have an "Unknown" value when the retrieve is issued. The slot name may be specified as "object.property" or just "object", which is shorthand for "object.Value".
- The In and Link to fields should be left empty

- In the Object Properties column, specify the property slots which are to be written to the fields in the database. In the Database Fields column, specify the corresponding field which is to receive each property slot.
- The Create New Record option must be left unselected. Only grouped writes can be used to create records.

Related Topics

Cursor Slot Specification	Slot Specification for Writes
Query Write Operations	Access String
Database Editor Windows	Beginning Database Operations
Write Unknown	Ending Database Operations
Writing to Databases	

Also, look up individual arguments and your database type for more detailed information.

Begin - (@BEGIN)

Syntax

The formal syntax of the begin statement is:

```
@BEGIN=quoted_string;
```

Note: When editing the Begin field in the retrieve or write dialog screens, do not enclose the entry in double quotes; the Rules Element will insert them.

Usage

The Begin argument is used in two different contexts:

- When using a relational database, the begin string will be sent to the DBMS server before the query string is executed. It is most frequently used to initiate write transactions. For example, an RDB transaction might be initiated by specifying:

```
@BEGIN="start_transaction read_write";
```

- A Sybase update transaction might be initiated by specifying:

```
@BEGIN="begin transaction change_price";
```

- The begin string can also be used to perform operations which are neither retrieve nor write requests. It can be used for operations such as deleting records, dropping or creating tables, and specifying a timeout period. Look up your database type for more examples on how this string can be used.
- When using SYLKDB spreadsheet databases, the begin string can be used to specify a database range name. If no range is specified, the Rules Element will use the default range Database. See the SYLKDB topic for further details.

The special construct `@V(obj.prop)` can be used in the begin field, as well as `@SELF`, and `@PROP` when initiating retrieve or write from a method.

For sequential queries the begin statement is performed only once, before the retrieval of the first record.

Related Topics

Interpretations @V(...)
Dynamic Values
Beginning Database Operations

Also, look up your database type for more detailed information.

Beginning Database Operations

General

Before beginning a retrieve or write operation, the Rules Element executes whatever statements have been included in the Begin field of the retrieve or write window.

For the SYLKDB database type, the Begin field holds the database range name. See the topic "SYLKDB" for more details.

For most relational databases, this field is not required. Some databases, such as Sybase, require that you include a Begin Transaction or similar statement in this field.

Actually, any valid SQL statement can be included in the Begin field since it is passed "as-is" to the database manager. This is useful for executing SQL DML (or data manipulation language) statements before retrieve or write operations. For example, statements like CREATE TABLE, DROP TABLE, and DELETE can be executed from the Begin field.

Note that the Rules Element doesn't make any effort to receive data from the statement in the Begin field, so coding a SELECT would not be very useful.

For example, to delete all the records from the CARS table before beginning a Write operation, a statement like the following could be included in the Begin field:

```
DELETE FROM CARS
```

Multiple statements can be included in the Begin field by separating them with semicolons (";").

If the statements executed from the Begin field fail, the retrieve or write operation will be terminated.

Note that for sequential retrieve operations, the Begin field is ONLY executed before the first retrieve--it's not re-executed for each record.

Specification

To fill in the Begin field you just include the SQL statements you would like executed before beginning the database operation. More than one statement can be executed by separating the statements with semicolons.

Interpretations ("@V(...)") can be used in the Begin field.

The following example shows a write operation using the Begin field to delete all the records from the CARS table before beginning the write.

The screenshot shows a dialog box titled "Database Retrieve". It has several input fields and a table. The "Begin" field contains "DELETE FROM CARS", "Query" contains "CARS", and "End" contains "COMMIT". The "Database Type" dropdown is set to "Oracle 7 Driver". The "Name" field contains "IDB_CAR_NAME!" and the "Link To" field contains "CARS_CLASS". There are checkboxes for "Create Object" (checked) and "Retrieve Unknown" (unchecked). Below these are radio buttons for "Always Forward", "Current Forward" (selected), and "Do Not Forward". At the bottom are "OK" and "Cancel" buttons. A table at the bottom shows "Database Fields" and "Object Properties" tabs. The table has two columns: "DB_PRICE" and "Price". The first row is highlighted.

Database Fields	Object Properties
DB_PRICE	Price

Figure 3-2 Using the Begin Field to Delete Records

Related Topics

Interpretations @V(...)

Dynamic Values

Begin - (@Begin)

File Retrieves @F(...)

Also, look up your database type for more detailed information.

Create New Record - (@FILL)

Usage

The Create New Record setting is only meaningful in the context of a grouped write only. Create New Record specifies whether new records may be added to a database during a grouped write. The system first updates existing records before creating new ones. If you already know that the records do not exist, you can instead specify the Insert Only setting so no update is attempted first.

In the write dialog screen this setting can be specified by clicking in the Create New Record check box. In a text format knowledge base it will appear as:

```
@FILL=ADD;
```

When New File is selected, Create New Record is automatically implied. The Insert Only setting is not compatible with either of these settings.

Related Topics

Grouped Write	Arguments Overview
Database Editor Windows	New File
Writing to Databases	Insert Only
Spreadsheets	

Create Object - (@FILL)**Usage**

The Create Object setting is used in the context of a grouped retrieve only. It controls whether or not dynamic objects are created during a retrieve operation.

In the retrieve dialog screen this setting can be specified by clicking in the Create Object check box. In a text format knowledge base it will appear as:

```
@FILL=ADD ;
```

If this setting is disabled, @FILL will not appear in the text format knowledge base.

Related Topics

Grouped Retrieve	Database Editor Windows
Arguments Overview	Retrieving from Databases
Debugging Operations	

Cursor Slot Specification**Purpose**

The Rules Element uses a cursor to determine the type of database transaction being requested. For sequential queries the cursor keeps track of the last record retrieved.

The presence or absence of a cursor determines whether the transaction is a grouped transaction:

Cursor absent Grouped transaction

Cursor present Atomic or Sequential transaction

If present, the value of the cursor slot immediately before the transaction determines whether an atomic or sequential query is being requested:

- If the value of the cursor slot is NON-NEGATIVE when a retrieve query is requested, the Rules Element treats the transaction as a sequential one.
- If the value of the cursor slot is UNKNOWN when a retrieve or write query is requested, the Rules Element treats the transaction as an atomic one.

- If the value of the cursor slot is NEGATIVE or NOTKNOWN when a retrieve or write query is requested, the Rules Element generates an error message.

Typically, in the case of a sequential retrieve, once a record has been retrieved a set of rules is fired to analyze the retrieved data. The cursor slot is used to hold the current state of the transaction so that the Rules Element knows how to resume its operation when another retrieve is executed to fetch the next record.

Value

The value of the cursor slot has different meanings depending on the type of database being accessed:

- For flat-file databases, the cursor holds the index of the last record retrieved, and is incremented each time a new record is retrieved. If the initial value of the cursor slot is 0, the retrieve will begin with the first record in the file. By specifying a positive cursor slot value, the retrieve can be started anywhere in the file. The cursor slot value can also be changed by rules to skip records in the file.
- For relational databases, the cursor slot holds a stream number (RDB) or an SQL cursor number (Oracle, Sybase, Ingres, ...). It is not modified when subsequent records are retrieved because the index in the virtual table is maintained internally by the DBMS. If several sequential transactions are active simultaneously, a unique cursor must exist for each one. For relational databases, the cursor slot must be initialized to 0 for any sequential transaction. During the first retrieve, the cursor slot will always be set to a positive value which will not be modified by subsequent retrieves (except when the retrieve fails because of an error or when all of the records have been retrieved). Consequently, the cursor slot value must never be modified by rules which are fired between retrieve transactions

When the retrieve encounters the end of a flat-file or the end of a virtual table of records (end-of-fetch), the Rules Element will set the value of the cursor slot to -1. The looping logic driving the application should test for this value and exit the retrieve loop.

The cursor will also be set to -1 if a query cannot be processed successfully for other reasons (data file not found, invalid field names, etc.). An error message will also be written into the transcript window.

Sequential Operations

With relational databases, sequential writes are usually performed in conjunction with sequential retrieves. A sequential write should use the same cursor as its associated sequential retrieve to ensure that the last record retrieved is updated. A sequential write does not modify the value of the cursor slot.

With flat-file databases, a sequential write can be executed independently of a sequential retrieve. In this case, the cursor value will directly index the record to be updated, and will be incremented automatically.

Related Topics

Atomic Retrieve	Atomic Write
Sequential Retrieve	Sequential Write
Cursor	Record Specification for Writes

Also, look up your database type for details about how the cursor slot should be specified.

Cursor - (@CURSOR)**Usage**

The cursor argument is only used in atomic and sequential transactions. If this argument is omitted the query is evaluated as a grouped query.

For sequential queries the cursor keeps track of the last record retrieved.

For atomic queries the value of the cursor slot indicates whether the query was successful.

In the retrieve or write dialog screens it is specified in the Cursor field. In a text format knowledge base it will appear as:

```
@CURSOR=slot;
```

The cursor is an integer object slot typically defined as `object.prop`.

Examples:

- `CurrentRecord.number`
- `TheCursor` (shorthand for `TheCursor.Value`)

The data type of a cursor slot must be integer.

The Cursor Slot Specification topic explains how the cursor is used in database transactions.

Related Topics

Atomic Retrieve	Atomic Write
Sequential Retrieve	Sequential Write
Cursor Slot Specification	Database Editor Windows
Arguments Overview	Record Specification for Writes

Database Interface Concepts**General**

The Rules Element database interface is used to transfer data between external data sources and the Rules Element's object representation. In many applications, the data is stored in an external file or database, where its format is very different from the Rules Element's object representation. The object representation - classes, objects, properties, and slots - is a structure for data which the inference engine reasons over. The database interface transforms and translates the data between its external format (a file or database) and the Rules Element object representation.

From another perspective, the database interface allows one to manage knowledge and facts separately in a Rules Element application:

- Knowledge is represented by rules describing the reasoning process, and a set of classes, objects, and properties which represent the world upon which the reasoning takes place.
- Knowledge is input by the application designers as they build the Rules Element application.
- Facts represent the actual data which is being processed by the knowledge base, and is represented in the Rules Element's working memory by classes, objects, and properties.
- In some applications, all of the facts are input by the user, in others some or all of the facts are obtained from external files or databases. The reasoning process can also produce new or altered facts, which can in turn be saved on external files or databases.

The database interface provides for a clean separation of knowledge and facts: knowledge is stored in knowledge bases (or KBs), facts are stored in external files or databases. The database interface allows the application to Retrieve data - or facts - from an external file or database, and Write the results of its reasoning - new or altered facts - into an external file or database. This approach has several advantages:

- The size of the knowledge base remains reasonable because only the rules and the structural representation of the facts (classes, objects, etc) are saved as knowledge. The facts or data are stored and retrieved separately.
- The size of the data or facts may be very large and is managed (in the case of relational databases) by powerful database managers which provide services such as data integrity for shared data, fast indexing, and so forth.
- Since they are stored in external data files or databases, data or facts can be accessed or produced by other applications.

Features of the Database Interface

The Rules Element database interface has many features which make it a good method for transferring data between the Rules Element and external files or databases:

- The database interface is invoked using Rules Element rules or methods. The database interface takes care of translating the parameters on the Retrieve or Write statement into the appropriate database access commands.
- Translation of data between the data's external representation (records, rows, cells, etc) and the Rules Element's object representation is handled automatically by the database interface.
- During retrieve operations, you can control whether the database interface should update existing Rules Element objects, or create new objects to represent the external data. Likewise, during write operations, the database interface can handle either updating existing records or creating new ones.
- Since Rules Element knowledge bases are portable, the Rules Element Retrieve and Write statements are also portable. Of course, the

portability of the application will be influenced by the portability of the target databases(s). Applications which use platform specific databases like ODBC will be less portable than those which use portable databases like Oracle or flat files.

- If the database changes, at the most only small modifications will have to be made to the Rules Element knowledge base. For example, changing a knowledge base to access a relational database instead of a spreadsheet file requires only a few parameter changes in the Rules Element knowledge base.

Of course, if by chance your database type is not supported by the Rules Element, there are other methods for interfacing the Rules Element to external files or databases. These include the following.

- A program using the Rules Element Application Programming Interface (API) could load a Rules Element knowledge base, read the records from the database, and volunteer the information into the Rules Element's working memory. When the inferencing process was complete, the program could use the Rules Element API again to extract the information from the Rules Element's working memory and write it back out to the database.
- A Rules Element Execute handler could be written, which is invoked via an "execute" statement in a Rules Element rule or method. When called, the handler would read data from the database and volunteer it into the Rules Element's working memory. The knowledge base can pass a list of the objects to receive data from the database to the handler in the "execute" statement. Another execute could be used to write the data from the Rules Element's working memory to the external file. Since an execute handler only receives object identifiers, or Atomids, the handler would still have to use the Rules Element API to extract the actual data from the Rules Element's working memory.
- A program could also be written as a Rules Element "question handler" to retrieve the data from the database. In this case, it would still have to use the Rules Element API to volunteer the data into the Rules Element, and would have the additional problem of determining whether the Atomid passed to the question handler should even come from a database (it could come from the user, or another data source).

If you use one of these methods for accessing your database, keep the following considerations in mind:

- The programs or handlers described are written in a high level language which supports the Rules Element API.
- The program or handler must do the transformation between the database or file's format and the Rules Element's object representation.
- The programs or handler is responsible for all interaction with the database or file's access methods.
- If your Rules Element application is to be portable, special care must be taken to ensure that the programs or handlers used to access the file or database are also portable.

When possible, using the Rules Element database interface to access external data bases and files is much easier than writing your own program(s) to handle the transfer. Only in the rare occurrence where the Rules Element doesn't support your external file type should it be necessary

for you to provide your own access. Refer to the Rules Element API Programmer's Reference Manual for details about the previously mentioned handlers needed to interface the Rules Element with unsupported databases.

Using the Rules Element Database Interface

Following are examples of applications which use the Rules Element database interface. In each case, one or more of the rules use the database interface to either get data from a database into Rules Element's objects, or take Rules Element objects and write them to a database:

Retrieving Records Sequentially

Assume that you have a Rules Element knowledge base to evaluate credit applications to determine whether or not the applications should be approved. The current application being processed is represented in the Rules Element by the object `current_application`, with the properties `applicant_name`, `income`, `prior_bankruptcy`, and `application_approved`. Rules will evaluate whether or not the application should be approved or denied, and the object's `application_approved` property updated as appropriate.

The credit applications themselves are stored in a relational database as rows in a table - a format which is very different from the Rules Element's class and object organization.

To implement this, a rule would use the database interface to retrieve the rows one at a time from the table. As each row is fetched, its column values are "pasted" into the properties of the object `current_application`. The Rules Element then uses the rules created by the application designer to determine whether or not to approve the application, and sets the `application_approved` property appropriately. Another rule would use the database interface to write the object out to the appropriate row in the database. During the write, the database interface will transform the object's properties into the appropriate columns in the row.

Retrieving Records as a Group

Another example is a knowledge base to assist in projecting the budget for a company which is divided up into departments. The example company is represented by objects in the class `department`, each of which has the properties `personnel_cost`, `overhead`, `rent`, `income`, `department_name`, and `final_budget`. The rules evaluate the needs of all the departments together, and update each department object's properties to reflect the final budget allocation.

The departmental information is stored in an EXCEL spreadsheet file. The file's format is very different from the knowledge base's representation of the data - it is organized in cells whereas the data in the knowledge base is organized in classes, objects, and properties. Another characteristic of this example is the requirement to process all the departments at once - in a group as it were.

Here, a Rules Element rule uses the database interface to read in all the department records from the spreadsheet, creating an object for each department. The objects are created in the `department` class, and the

database interface takes care of pasting the appropriate cells into each object's properties. The rules then develop a proposed budget and update each object's `final_budget` property in the Rules Element's working memory. Another rule writes the updated objects out to the EXCEL spreadsheet - with the database interface transforming the objects and properties back into EXCEL's cell-type organization.

Retrieving One Record at a Time

A system for configuring automobiles accepts input from a car buyer on the features they would like to order with their car. Each feature is represented by an object with the properties `Feature_Name`, `Color`, `Style`, `Price`, `Dimension`, and `Stock_Number`. The user only selects the `Feature_name`, `color` and `style` for each feature - the remaining information - `Price`, `Dimension`, and `Stock_Number` - must be retrieved from a database. The records are accessed one at a time, as the features are selected.

In this knowledge base, a rule would use the database interface to retrieve the appropriate record from the database as each feature was selected by the user. As each record is retrieved, the Rules Element would update the object's `Price`, `Dimension`, and `Stock_Number` properties with the information from the database. The rest of the knowledge base evaluates the feature's compatibility with other features (represented by previously created objects) already on the car.

Summary

You can see that the database interface is very much like a "pipe" between the Rules Element's working memory and an external data source like a database or a spreadsheet. However, the database interface does much more than simply transfer data - it also transforms it between the Rules Element's class-object-property representation and the external data source's format. You can also see that the database interface is capable of different types of processing - retrieving all the records one at a time, retrieving all records at once, and retrieving only one record. See the Related Topics list for more information about these operations.

All of these examples use only one type of database, but it's possible to read and write multiple database types from the same Rules Element knowledge base. Since the database interface always reads into and writes from the Rules Element objects, the Rules Element application doesn't have to be concerned about conversions between different database types.

Related Topics

Databases	Spreadsheets
Retrieving from Databases	Writing to Databases
Database Editor Windows	Arguments Overview

Database Editor Windows

Usage

The retrieve and write dialog windows behave in a manner similar to that of the other Rules Element editors like the Rule Editor and the Object Editor. A cell can be selected by clicking on it with a mouse, or by using the RETURN, TAB or DOWN ARROW keys to move forward through the fields, or by using the UP ARROW or shift-TAB keys to move backwards through the fields.

Figure 3-3 Retrieve Dialog Screen

Figure 3-4 Write Dialog Screen

A DBMS can be selected by clicking on the desired name in the list of database types.

The Copy property pop-up menus can be used while editing the Object Properties list to avoid typing errors in atom names.

Note: : When editing the fields Begin, Query and End, do not add double quotes. They will be inserted automatically by the Rules Element.

Each field and button in the write and retrieve windows has an associated key word. The correspondence between the two is as follows:

Begin	@BEGIN
Query (1st part of query cell)	@QUERY
Query arguments (2nd part of query cell)	@ARGS
End	@END
Name	@NAME
Cursor	@CURS
In	@ATOMS
Link to	@CREATE
Database Fields	@FIELDS
Object Properties	@PROPS or @SLOTS
Database Type	@TYPE
Create Object (Retrieve)	@FILL
Create New Record (Write)	@FILL
Insert Only (Write)	@FILL
New File (Write)	@FILL
SqlError (Write)	@ERROR
Retrieve Unknown (Retrieve)	@UNKNOWN
Write Unknown (Write)	@UNKNOWN
Forward buttons	@FWRD

Saving Fields

When you are in one of the Database Editor windows and you click on the OK button, the arguments you have entered are saved in the Rule or Method Editor, prefixed by their keywords (see above). The entire argument list of your Retrieve or Write can be viewed in the edit line at the top of the Rule or Method Editor. To do this, you can either clear the Retrieve from the first column and then click in the third column, or you can click on the right side of the third column to bring up the pop-up, and then move away from it, leaving the argument list displayed in the edit line. A knowledge base saved in a text format can be edited with any standard text editor.

Related Topics

Arguments Overview	Write Operator
Retrieve Operator	Retrieving from Databases
Writing to Databases	Databases
Spreadsheets	Database Type
Debugging Operations	

Also, look up individual arguments and your database type for more detailed information about completing the Database Editor windows.

Database Type - (@TYPE)**Purpose**

This keyword specifies the type of database to be accessed. It can take one of the following values:

DAL_CL1	Apple's Data Access Language (formerly CL/1)
DB2	IBM's DB2 relational database
DBF3	dBase III
INFORMIX	Informix's SQL relational database
INGRES	Ingres' SQL relational database
NONSTOP	Tandem's relational database
NXP	Rules Element's spreadsheet
NXPDB	Rules Element's database table
ORACLE	Oracle's SQL relational database
RDB	DEC RDB relational database (VAX/VMS only)
RDBCDD	RDB installed with the CDD common dictionary
SQLBASE	GUPTA's relational database
SQLDS	IBM's SQL/DS relational database
SQLSERVER	Sybase's OS/2 relational database
SYBASE	Sybase's SQL relational database
SYLK	EXCEL spreadsheet
SYLKDB	EXCEL database
VAX SQL	Interface to RDB/VMS for DEC
WKS	Lotus 1-2-3 spreadsheet
WKSDB	Lotus 1-2-3 database

Note: : This list is continually growing and additional database interfaces may be available that are not documented here. Contact Neuron Data to determine the availability of any database interface not listed above.

SQLSERVER is not supported on the PC.

NXP, NXPDB, SYLK, SYLKDB, WKS, WKSDB, and DBF3 are available on all versions of the Rules Element, even if the spreadsheet or database application is not available for that platform. These

formats are provided to ensure compatibility across platforms. For example, a flat-file database created by dBase III on an IBM-PC can be read by the Rules Element database interface on a VAX or UNIX platform.

When opening flat-file databases, the Rules Element checks the file header and will generate runtime errors if there is a mismatch between the database type specified and the file header.

Related Topics

Database Editor Windows	Arguments Overview
Databases	Spreadsheets
DBF3	INGRES
ORACLE	RDBINFORMIX
SYBASE	WKS
SYLK	VAX SQL
Databases	

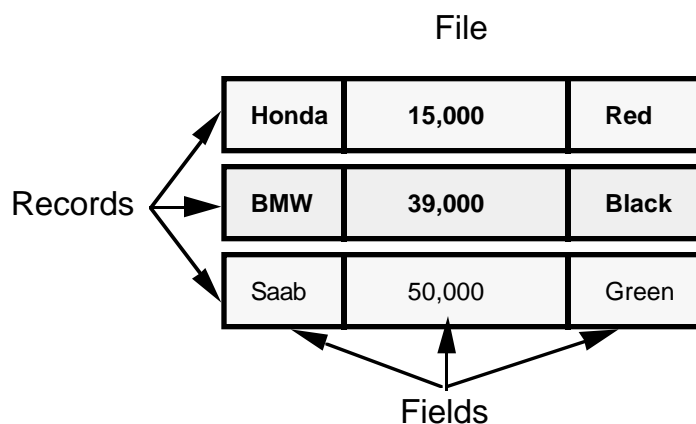
The concept of databases describes a much more typical organization of data which is common to all other database and file formats supported by the Rules Element. Although the terminology varies widely among the file types and products, the basic data structure is the same.

Terminology

In a database, data is grouped into logical entities which we will refer to as records. A record represents an individual thing such as a transaction, an inventory item, an event record, or a personnel record. The decision of what goes into a record is completely up to the application designer.

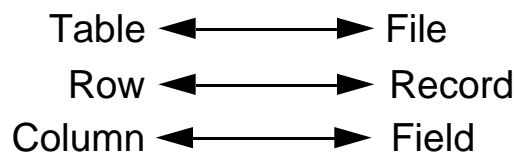
Each record is divided up into fields, which represent individual data items about the thing the record represents. For example, a car inventory record could contain fields for the car's price and its model. Generally, records of the same type contain the same fields, but this is not necessarily so.

Records of the same type are grouped together into files. Again, generally all the records in a file contain the same fields, but some file or database formats allow some fields to be omitted in some records.



File formats like NXPDB, DBASE III, RDB, and others use the terms file, record, and field to describe this organization. Relational databases such as

ORACLE, INGRES, SYBASE, and Apple's DAL use the terms table, row, and column, but the structure is exactly the same: tables (files) are composed of rows (records), which in turn are made up of columns (fields).



With relational databases, it is important to note that it is possible to access rows from two or more tables in a single request using a JOIN. Nonetheless, the Rules Element database interface always sees a single virtual table which is the result of the join operation. No matter how many tables are involved, the data is still presented to the Rules Element database interface as a collection of columns, organized into rows, from a single table.

Related Topics

Database Type	DBF3
INGRES	Oracle
RDB	SYBASESYLK
WKS	Spreadsheets
Retrieving from Databases	
Writing to Databases	

DBF3

General

DBF3 is the dBase III format. The Rules Element can read and write this format on any platform, even if the data file can be used directly only by dBase III on the IBM-PC.

Header names cannot exceed 10 characters according to DBF3 specification. By default, new DBF3 files are created with the following field widths:

boolean	1 (Logical value)
integer	10
float	10 + <Current Precision in the Rules Element>
string, date, time	30
special property Value	30

The Rules Element Flat-File Format topic explains how you can define your own field width and override these default values.

In dBase III, boolean values are stored in a one-character field. By default, the Rules Element uses the following formats for boolean values:

When writing into the DBF3 file

TRUE	becomes y
FALSE	becomes n
NOTKNOWN	becomes *
UNKNOWN	becomes ? when Write UnKnown is enabled, (otherwise nothing is written).

When reading a DBF3 file

y,Y,t,T	are interpreted as	TRUE
n,N,f,F	are interpreted as	FALSE
u,U,?	are interpreted as	UNKNOWN
*	is interpreted a	NOTKNOWN

Note that data and time data types, as well as indices are not supported.

Related Topics

Databases
Retrieving from Databases
Writing to Databases
Rules Element Flat-File Format

Debugging Operations

This section contains information that might be of use when it comes to debugging why your Retrieve or Write operation is not behaving exactly as you had wanted. Among the topics covered are: using the Transcript window, stand-alone query testing, and miscellaneous commonly occurring errors.

Transcript Window

The Transcript window is probably the single most useful debugging tool for debugging database interface problems. To cause the Rules Element to write to the Transcript window when in the Development interface, you should select the “Enable Write” option from the window’s popup menu.

Transcript Window Usage

When trying to debug a Retrieve or Write using the Transcript window, it is a good idea to try to come up with a test case with only one or at most a few rules. This avoids filling the transcript log with volumes of information not relevant to the problem, and also makes the debugging go faster since the entire KB is not being run. If this is not possible, a couple of other possibilities exist. One possibility is to set the Rules Element breakpoints before and after the database operation. You can enable the Transcript window when the first breakpoint is reached, and disable it afterwards. Another possibility is to add new conditions or actions just before and after the database operation to enable and disable the Transcript writing.

Even in this case, a Retrieve and/or Write can still generate a significant amount of information to the Transcript window. If this slows down the application too much, you can select the Close option from the Transcript window's popup menu while the writing is going on. This should cause the application to record the information, but run significantly faster, since the window is not being updated (typically not a fast operation). When the session ends, you can then bring the Transcript window up to browse through the information reported by the Rules Element.

Database Messages

The Transcript window should now contain messages of the form: "xxx Interface executing: ..." (or something along these lines), where "xxx" is the specific database interface you have (e.g. Oracle, Sybase, etc). You should try to find each occurrence of one of these lines and determine if they were successfully executed. Any failure should be apparent by the appearance of an error message following the executing message. These error messages will usually have been generated by the specific database server, returned to the Rules Element, and displayed in the Transcript window by the Rules Element. Note that not all error messages are fatal: (e.g. warnings about trying to drop tables that don't exist, etc). Other messages, however, will be fatal (e.g. access failed because of invalid database access string, and field name doesn't exist).

Error Slot

If you want to trap each error and test the value before proceeding, you can use the SqlError field of the Database Editor window to create an error slot. The error slot you specify will receive the error message or number generated by the specific relational database server. If the database returns either an error number or an error message at runtime, the transaction is immediately halted, and the inference engine automatically sets the left-hand side Retrieve or Write condition to FALSE. If no error slot is specified, error messages that are generated at runtime can be viewed in the Transcript window that you enable.

Query Syntax

You should also check that the "..." part of the executing message appears to be valid query syntax (SQL, RDO or whatever) for your database. In many cases, you can execute almost exactly the same query outside the Rules Element environment by using an interface provided for the database. For example, with RDB, you could use the RDO interface; with ORACLE, you could use SQL*Plus; with SYBASE, you could use isql and so on.

Sometimes the problem may be with the presence (or absence) of quotes around the information being passed to the database. The Rules Element normally knows to transfer integer fields without quotes, string fields with quotes, and so on (although this is not always the case). The Rules Element does not normally know the datatype of the database field it is writing into and might inappropriately provide (or not provide) quotes around the database field value. This can be detected by the Transcript window error message, and confirmed by a standalone query test.

In most places where quotes are required (in the Name field and in the Fields list) it's possible to provide a "hint" to the Rules Element to override

its default handling. The way to do this is to preface the database fieldname with "{I}" (denoting integer-like, or more generally, numeric) or "{S}" (denoting string, the normal default). Specifying something as a numeric fieldname should force the Rules Element to omit putting quotes around the field value. Likewise, specifying "{S}" should force quotes to be placed around the field value. This syntax is documented under the Query Language topic, and the datatypes / database interfaces that need this syntax are documented under specific database types.

Rules Element Messages

The Transcript window could also indicate that the problem with the database operation is not with the database server, but on the Rules Element end of the transaction. For example, the field might exist, but there is no object available to Retrieve the information into. Similarly on a Write, there might not be an object from which to obtain information, or the Name field might be causing the wrong database record to be updated.

A more common problem occurs with formats being incompatible between the Rules Element default and the default for the appropriate database. See your database topic in this chapter for a datatype compatibility table. Typically this occurs with dates and times. The Transcript window again should show the information being returned from the database and the Rules Element format(s) which are being tried for a match.

Other Errors

A variety of other common errors may occur as follows.

Name Field

In a Write statement, you should never use a database field name in both the Name and Field areas. This might work in some cases, but in other cases it will lead to unpredictable results. It is acceptable to duplicate the database field name in this manner in a Retrieve statement, however.

The Name field consists of a series of expressions like: 'root1!field1!root2!field2! ... with a maximum of five root/field combinations being allowed. On RETRIEVES and WRITES, typical field names are strings and integers. Some other conversions may be done by the Rules Element to retrieve into a proper object name, but the conversion is not always reversed on a write operation.

Commit

Following a write operation or on the last write before the end of the session, you should typically specify "commit" or "rollback" in the End field. You should issue a "commit" if you are satisfied with what has been written; "rollback" otherwise. The Rules Element does NOT automatically commit for you (this would negate the advantages offered with commit/rollback). However, you should be aware that when you do a RESTART, the Rules Element automatically does a rollback. If you forget the commit, even though all your database writes succeeded, the actions from this session will be totally undone by the rollback.

No Fields Specified

A common mistake when coding a Grouped Retrieve is to omit the Fields and Props from the retrieve, thinking this will retrieve all the properties of the object(s) you are interested in. This, however, causes the Rules Element to construct a query which attempts to select all the properties which the Rules Element knows about from the database/table (using the property names as field names). Even if the knowledge base is carefully constructed to only include properties known to be present in the table, the Rules Element has special properties (e.g. "Value") which probably won't be in the table and will cause the query to fail. This should be noticeable if, again, you look at the Transcript output and notice the names of the various database fields that the Rules Element is trying to retrieve information from.

Create Object Not Specified

Another common mistake when coding a Grouped Retrieve is to neglect to check the Create Object (@FILL=ADD) box in the Database Editor window. In this case, the Rules Element only retrieves those rows whose names match the names of existing objects as specified by the Name field. If no Name field is specified, then the Rules Element uses a Name field in the database to get an object name, and tries to find an existing object with that name. If the field doesn't exist in the database, or the field exists, but there is no object with that name, the Retrieve will succeed from the Rules Element's perspective, but fail from yours. If a Name field has been specified with roots and database field names, then the Rules Element will look for an existing object with that name. Again, if it doesn't exist, the Retrieve will not return any information from the database.

Related Topics

Query Language	Name
End	Create Object
Retrieving from Databases	Writing to Databases
Database Editor Windows	Formats
Existence Filtering	SqlError

Dynamic Values

You can use reserved words or arguments to tailor your query so that values in the query are not determined until the query is evaluated.

Using Reserved Words

You can use two reserved words to tailor your queries:

- Use @V to use the current value of the property slot.
- Use @SELF to use the current object whose property is being evaluated. This is valid only when the Retrieve or Write is in an Order of Sources or If Change.

For example, this query finds the value of `MyFavoriteColor.value`. If the value is blue, then the query retrieves all records where the color is blue:

```
cars where color contains @V(MyFavoriteColor.value)
```

This example uses the value of the current object to find all employees whose salary is greater than that value:

```
employees where salary > @V(@SELF.amount)
```

Using Arguments

You can also use arguments to tailor your queries. To use arguments, specify an identifier in the query. The Rules Element then checks the contents of the Query Arguments field (second cell) to determine the value of the identifier.

SQL identifiers use this format:

```
:argument
```

where `argument` is a string that you supply.

RDB identifiers use this format:

```
!argument
```

where `argument` is a string that you supply.

This is an example of a query that uses an argument:

```
employees where salary > :v1
```

:v1 refers to the first value in the Query Arguments field. The Query Arguments field contains this:

```
TooBigSalary.amount
```

Related Topics

Query Retrieve Operations
Writing to Databases
Query Language
Query Arguments

Query Write Operations
Database Editor Windows
Interpretations @V(...)

End - (@END)

Syntax

The formal syntax of the end statement is:

```
@END=quoted_string;
```

When editing the `end` field in the retrieve or write dialog screens, do not enclose the entry in double quotes; the Rules Element will insert them.

Usage

This argument is used only with relational databases. It contains a statement which will be sent to the DBMS server just before the resources involved in the transaction are released (after the last record has been retrieved or updated). Typically, this string will contain Commit or Rollback statements.

The special constructs `@V(obj.prop)`, `@SELF`, and `@PROP` can be used in the End field.

Note: For sequential queries the end statement is performed only once, after the retrieval of the last record.

Related Topics

Database Editor Windows
Arguments Overview
Interpretations @V(...)

Debugging Operations
Ending Database Operations
Dynamic Values

Ending Database Operations

Purpose

Once the database successfully completes a retrieve or write operation, it executes whatever statement has been included in the End field of the database retrieve or write window.

The End field is used **ONLY** with relational databases such as Oracle, Informix, Sybase and INGRES. For most databases, this field is only used for write operations, and in these cases will contain a SQL COMMIT or COMMIT RELEASE statement.

The COMMIT statement is used to signal the database manager that all of the updates to the database are complete, and should be made permanent. Optionally, the word RELEASE can also be included to tell the Rules Element to close its connection with the database manager.

It's a good practice to always include a COMMIT in the End field, since different database managers have different default actions if an application terminates without issuing a COMMIT or ROLLBACK statement. Some databases will automatically commit the changes, other will assume that a failure has occurred and will roll the changes back.

Actually, any valid SQL statement can be included in the End field since it is passed as-is to the database manager. This is useful for executing SQL DML (or data manipulation language) statements after retrieve or write operations. For example, statements like CREATE TABLE, DROP TABLE, and DELETE can be executed from the End field.

Note that the Rules Element doesn't try to receive data from the statement in the End field, so coding a SELECT wouldn't make much sense.

Multiple statements can be included in the End field by separating them with semicolons (;).

Specification

Filling in the End field is quite simple--you just include the SQL statements you would like executed when the database operation is complete. The key words COMMIT and ROLLBACK have been defined by a resource file with the definition supported for each database. When the Rules Element parses the End field and finds the keyword COMMIT or ROLLBACK, it will convert it to the correct database SQL query. The resource file ensures that these keywords can be used without regard to which database system is to be accessed.

More than one statement can be executed by separating the statements with semicolons. Interpretations @V(...) can be used in the End field.

The syntax of the COMMIT and ROLLBACK keywords for your database are defined by the resource file `nxda.dat`. You can view and edit the definitions in the Resource Browser by displaying the resource names `NxDa.Commit.DbName` and `NxDa.Rollback.DbName` (where `DbName` is the name of the database supported or ANSI, in the case of the default ANSI SQL). The Rules Element will try to execute the query as defined in the resource file, if it doesn't find a resource defined for a particular database, it will use the ANSI SQL type.

The following example shows a write operation using the End field to commit the changes after a write operation.

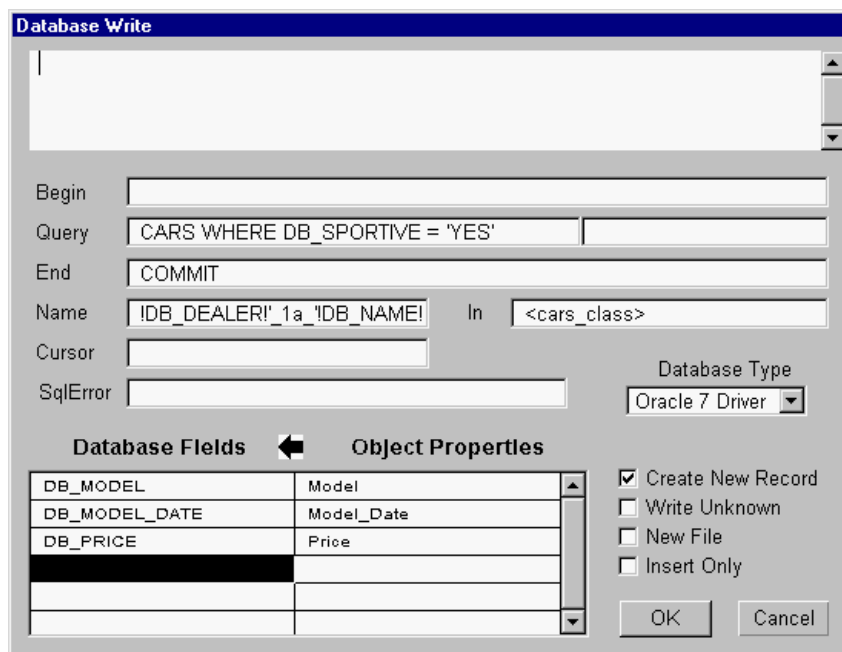


Figure 3-5 Filling in the End Field

Related Topics

Database Editor Windows
Arguments Overview
File Retrieves @F(...)
Dynamic Values

Debugging Operations
End
Interpretations @V(...)

Existence Filtering Operations

Existence filtering is used in grouped retrieve operations which specify specific objects to update. During the retrieve operation the Rules Element determines if the specified objects exist to hold the record's fields. It is possible to use the object's existence as a final criteria for determining if the record should be retrieved or not.

Existence filtering cannot be used with sequential and atomic retrievals because the slot names (object.property combinations) are always specified explicitly for these operations. The Rules Element's rule compiler requires that explicitly named slots exist when the rule is compiled, thus the object will always exist when the retrieve is executed.

Existence filtering can be used with grouped retrieval since the object may not exist when the retrieve is executed. Existence filtering can be used to bypass retrieving a record if the object doesn't already exist, or if the object doesn't already exist in the In list. This section discusses how to use existence filtering.

Usage

During a retrieve, the Rules Element builds an object name to identify which object will hold the current record's fields. Existence filtering can then be used to make the final determination of whether or not to retrieve the record. Stated simply, existence filtering means:

- If the object doesn't already exist, then don't create it (and bypass the record).
- If the object isn't in the In list - a list of eligible objects and classes - then bypass the record.

Existence filtering is a good way to update an existing set of objects from a database, reading only those records which correspond to existing objects.

For example, assume that there are two objects in the Rules Element's working memory - `car_1` and `car_2`, and that the `CARS` table contains ten records for `car_1` thru `car_10`. In order to fill in the property slots of `car_1` and `car_2`, all of the `CAR` records could be retrieved, but this would have the undesirable side affect of creating objects for `car_3`, `car_4`, etc. By using existence filtering, ONLY the fields from the `car_1` and `car_2` records could be retrieved, and the rest of the records bypassed (since no objects exist to hold the record's fields).

Existence filtering can work on two levels--it can test to see if the object exists anywhere in the Rules Element's working memory, or it can test to see if an object is the member of an In list.

Check Memory

To test for object existence in all of the Rules Element's working memory, the Create Objects box must NOT be checked in the Retrieve window. If this box is not checked and the In field is empty, then the Rules Element will look through all of its working memory for a matching object.

Check In List

The In list field can be used to restrict the search for a matching object to a specific set of object names and/or class specifications. Both object names and class specifications can appear in the In List.

Object names are used "as-is" by the Rules Element. The Rules Element compares the object name generated for the record to the object names in the list. If it's in the list, the record is considered to have passed the existence test.

Specify a class name by enclosing it in angle brackets. For example, to match the objects in `car_class` against the generated object name, `<car_class>` should be specified in the In list.

Actually, `<class_name>` is an existential pattern matching operation with no test, therefore all objects currently in the pattern matching list are used. If this is the first time the particular `<class_name>` specification appears in the LHS or RHS of the rule, then all the objects in the class will be used. However, if previous pattern matching operations had trimmed the list, then only those objects remaining in the list are matched against.

Using the In List in this fashion is useful for limiting the records retrieved using a piece of data which is "known" to the Rules Element, but is not contained in the database being retrieved from.

For example, assume that the objects in the class `car_class` have a property `Color`, which does NOT have a corresponding field in the `CARS` database. Also, there are ten `car_class` objects (`car_1`, `car_2`, `car_3`, etc), and only two of them -- `car_1` and `car_5`--have a `Color` slot of `Red`.

To retrieve ONLY the records for Red cars would be difficult since the `CARS` database has no `Color` field to use as a reference. However, using pattern matching, `<car_class>` can be trimmed to contain only `car_1` and `car_5`, and existence filtering used to limit the records retrieved to only those objects left in the list. To do this, include a statement like the following in the LHS or RHS of the rule which issues the Retrieve:

```
=      <car_class>.Color      "Red"
```

This will cause `<car_class>` to yield only `car_1` and `car_5` the next time it is referenced in the LHS or RHS of the rule. By including `<car_class>` in the In list, records would only be retrieved for the objects in the list: `car_1` or `car_5`.

Specification

To use existence filtering, one or more of the following must be done when filling in the Retrieve window:

- Make sure the Create Objects box is NOT checked.
- To further restrict the search for an object, specify an In list of object and/or class specifications in the In field.

Do NOT check the Create Objects box and include names in the In field. This can have undesirable side effects such as creating "ghost" objects which are attached to no classes and have no properties.

Ensure Create Objects Box is NOT Checked

Checking the Create Objects box tells the Rules Element NOT to use existence filtering. This means that if the Rules Element does NOT find an object to match the generated object name, the Rules Element will create an object to hold the record's contents.

If Create Objects is NOT checked and nothing is specified in the In List (the In field in the Retrieve window), then the Rules Element will look at all objects in its working memory for a matching object.

Specifying an In List

To limit the search for a matching object to a specific set of objects, specify a list of object names and/or class names in the In field of the Retrieve window, separated by commas. The class names are specified as `<class_name>`, which is actually an existential pattern matching operation. Remember that if `<class_name>` has been used previously in the LHS or RHS of the rule, only those objects which passed the pattern matching operation will be in the list when it is used by the Retrieve operation.

Related Topics

Database Editor Windows	Debugging Operations
Arguments Overview	In Filtering List
Existence Filtering Example	In List
Grouped Retrieve	Create Object

Field Name Specification

Usage

To specify the field names to be retrieved or written, you fill in their names in the left hand side of the Fields and Properties List - the double column list box at the bottom of the Database Editor window. A field name may be specified more than once in the list.

Usually, the field name is specified as a simple name (such as `DB_MODEL`, `DB_PRICE`, etc.), but additional information may be included for some databases including field width or context names. Some relational databases allow you to specify an expression like `DB_PRICE*2` or `substr(DB_MODEL, 1, 7)` as a field name.

Related Topics

- Fields List
- Retrieving from Databases
- Writing to Databases
- Object Names In Retrieve Operations
- Database Editor Windows

For precise information on what is allowed for a given database type, look up your database type.

Fields List - (@FIELDS)

Usage

The fields list can be specified in all types of transactions, except when using spreadsheet files (NXP, SYLK and WKS). This list is edited under the heading `Database Fields`, in the left side of the double list box at the bottom of the retrieve and write dialog windows and is used to specify the mapping between database fields and property slots of Rules Element objects.

Additional information may be associated with each field name: in the case of data files, field width; in the case of RDB, context variable; in the case of some relational databases field names may be expressions. The precise syntax of field names is specific to a particular database and is described in more detail under specific database types.

In text knowledge bases, the field list is saved as a list of quoted strings. The formal syntax is:

```
@FIELDS=list of quoted_strings
```

Note: : When editing the fields list do not add double quotes. They will be inserted automatically by the Rules Element.

Related Topics

Databases
Database Editor Windows
Arguments Overview
Retrieving from Databases
Writing to Databases

For precise information on what is allowed for a given database type, look up your database type.

File Retrieves - @F(...)

Usage

Recall that the Rules Element's @F(filename) syntax allows to you provide the name of a file that is to be read into the Rules Element at that location. In the Rules Element's database interface, you can take advantage of this syntax in the BEGIN statement.

For example, you could have a BEGIN statement like:

```
@BEGIN= "@F(myfile.sql)";
```

that could contain some specific SQL statements pertaining to this operation. The file could contain a specific start string to allow a read/write transaction, or it could also contain SQL statements specific to dropping/creating tables and deleting records for example. Many of the examples provided with the Rules Element take advantage of this technique for dropping and creating tables. This allows the knowledge base to be relatively independent of the particular database interface, and the external file contains the SQL specific to the database being used (SQL implementations do vary from vendor to vendor). Some of the specific database administration and maintenance operations possible in the BEGIN statement are found under the Begin topic.

Note that you could combine @F and @V for additional flexibility by specifying something like: @F(@V(slot)), which would let you choose the file dynamically.

Related Topics

Database Editor Windows Interpretations @V(...)
Begin Dynamic Values
Beginning Database Operations

Formats

General

Formats are used to describe how the values contained in the database cells are mapped into the values of the Rules Element slots. These formats can be attached to properties or to individual slots.

Usually the mapping of values is straight-forward. For example, textual information is usually stored as character strings in the database and is transferred without modifications to Rules Element string slots. The mapping is less obvious in the case of dates, where different databases use different formats for representing dates. Local conventions may also affect the representation of dates.

Formats are described in more detail in Chapter One, “Application Development Features”, but the most important points are:

- The first or the first two formats specify how the values will be formatted for output. They define how values will be written to the database.
- All the formats may be used to interpret input values. The Rules Element tries to scan the incoming strings according to every format specified until a match is found. If the external string does not match any format, a warning message is displayed in the transcript and the slot is set to NOTKNOWN.
- If you start a format description with an exclamation mark (!), then the format will be ignored for database transactions.

Specific database type topics contain additional information on various formats that are required for specific databases to be able to properly retrieve various database datatypes into Rules Element slots. The following examples illustrate the important role formats can play in database transactions.

Example 1 Boolean format

```
Format = @N="*" ;@U="?" ;1;0;T;F;
```

In this case, a NOTKNOWN value will be written as an asterisk (*), an UNKNOWN value as a question mark (?), a TRUE value as a 1, and a FALSE value as a 0. A cell containing the single letter T will be interpreted as a TRUE value in a Retrieve operation, but the database cell will be updated with a 1 or 0 in a Write operation.

This format allows you to store boolean values in a single character field. An eight character field is required if you do not specify any format, because any NOTKNOWN values are written as the NOTKNOWN keyword.

With this format, values will also be displayed as single characters in the Object Network, reports, etc. You can avoid this and reserve single characters for database operations. If a format description starts with an exclamation mark (!), then the format will be ignored for database transactions. To keep the ability to store single characters in database cells, but display values as True and False, the above format could be changed to:

```
Format = !@N="NOTKNOWN" ;!@U="UNKNOWN" ;@N="*" ;@U="?" ;!True;!False;1;0;
```

Example 2 Integer format

```
Format = d*;
```

This format can be used when your data is stored as floating point data in the database, but you want to retrieve it in an integer Rules Element slot. The decimal part of the database data will be ignored (the value is truncated, not rounded to the nearest integer).

If you don't specify this integer format and your database data is formatted as floating point numbers, the Rules Element will not be able to interpret the database data and will set the slot values to NOTKNOWN. Of course, this problem can also be avoided by using the `float` datatype of the Rules Element.

Related Topics

Debugging Operations
String to Numeric Conversion
Retrieving from Databases
Writing to Databases

Forwarding Strategy - (@FWRD)

Purpose

The forward strategy setting is only used in retrieve operations. It specifies whether the passing of values to property slots during retrieve operations will cause the system to place hypotheses on the Rules Element agenda for evaluation

This setting is specified with the three check buttons Always Forward, Current Forward and Do Not Forward in the Retrieve dialog window. The corresponding values in text knowledge bases are as follows:

Always Forward	@FWRD=TRUE;
Do Not Forward	@FWRD=FALSE;
Current Forward	@FWRD string not specified

Always Forward specifies that database retrieves which affect the slot values of the LHS conditions of any rule will always cause those rules to be placed on the agenda for evaluation. Do Not Forward specifies that database retrieves will never cause rules to be placed on the agenda. Current Forward specifies that the forwarding strategy in effect when the retrieve is executed will be used to determine whether rules are placed on the agenda.

Related Topics

Database Editor Windows
Retrieving from Databases
Arguments Overview

Grouped Retrieve

General

Grouped retrieval can be used with both flat-file databases and relational databases.

A grouped retrieve operation reads multiple records in one operation. As the Rules Element processes each record, its fields are read into slots. All of the fields from a given record are read into the same object's slots--"transforming" the record-field relationship into an object-property relationship.

A typical use of Grouped retrieve is to propagate a Rules Element class with objects created from records in a database. The objects can then be used in the Rules Element rules just as any other objects would be. Another use of grouped retrieve is to update a set of objects from data in a database. In this case, only the records are retrieved which have corresponding objects in the Rules Element's working memory.

For example, a grouped retrieval could be used to read all the records from the CARS database into the Rules Element working memory, creating an object for each record and attaching it to the `cars_class` class.

Grouped retrieves don't require supporting logic in other rules to retrieve the records. However, the appropriate class and object definitions must exist so that the Rules Element has a model for transforming the records and fields into objects and slots (object.property combinations).

Specification

Grouped retrieves are recognized by the absence of a Cursor slot in the Retrieve window.

A grouped retrieve does not have to retrieve all the records from the database--in fact this is usually VERY undesirable since an object will probably need to be created for each record in the database. To limit the records retrieved, a query can be included to filter the records read. For relational databases, you can use any query accepted by the database manager (usually an ANSI SQL statement), for flat-file databases, you can use the Rules Element's SQL-like query language to filter the records.

A grouped retrieve can either update existing objects, or create new objects and attach them to one or more classes.

Another technique for filtering records to be retrieved is to qualify them based on whether or not a corresponding object already exists to hold the record's fields. The search for an existing object can be thru all of the Rules Element's working memory, or confined to a specific list of objects and classes.

Fields

To build a grouped retrieve, complete the Retrieve screen in the Database Editor window as follows.

- Specify Retrieve as the operator in the LHS or RHS of the rule.
- As the first operand of the Retrieve, specify the database access string if a relational database is being accessed. If a flat file database such as

NXPDB or DBASE III is being accessed, specify the file name. See the Access String Specification topic for more information.

- In the database Retrieve window, click on the appropriate selection in the Database Type field for the database being retrieved from.
- The Begin field should contain whatever is appropriate for your database. See the Beginning Database Operations topic for more information.
- For a relational database, specify the table name to be accessed in the Query field. If you want to limit the records retrieved by the retrieve, you can also include a SQL query (for relational databases) or a Rules Element SQL-like query (for flat file databases). See the Query Retrieve Operations topic for more information on the Query field.
- The End field should contain whatever is appropriate for your database to end a transaction.
- The Name field is used to construct the slot names (object.property combinations) into which the record fields will be read. The slot names are built dynamically using data from the record. See the Slot Specification for Retrieves topic for more information.
- The Cursor MUST be left empty
- The In field is used to specify a list of objects (and/or their classes) in which the object selected to hold the record's fields must exist in order for the record to be processed. See the Existence Filtering Operations topic for more information.
- If objects are to be created dynamically as the records are retrieved, the Link to field should contain the name(s) of the classes to which the new objects should be linked. The Create Record option must be selected if objects are to be created dynamically. The In field must not be used in this case to avoid creation of objects outside of the specified list.
- In the Database Fields column, specify the names of the database fields to be retrieved. In the corresponding Object Properties column entries, specify the property slots into which the fields should be retrieved. See the Slot Specification for Retrieves topic for more information.

Related Topics

Object Names In Retrieve Operations	Query Retrieve Operations
Database Editor Windows	Slot Specification for Retrieves
Retrieving from Databases	Link To
Name	Field Name Specification
In Filtering List	Existence Filtering Operations

Also, look up individual arguments and your database type for more detailed information.

Grouped Write

General

Grouped writes can be used with both flat-file databases and relational databases.

A grouped write will write multiple object's slots in one operation. All of the slots written to a given record come from the same object, transforming the Rules Element's object-property relationship to a record-field relationship in the database.

A typical use of grouped write is to write an entire class of objects out to a database. It's also possible to write out every object in a list, or every object in a list of classes, to the database.

For example, a grouped retrieval could be used to write all the objects from the `cars_class` into the `CARS` table, creating a row for each object in the `cars_class`. As each object is written, the appropriate slots (object.property combinations) from the objects are written into the columns of the new rows.

Grouped writes don't require supporting logic in other rules to write the records.

Specification

Grouped writes are recognized by the Cursor field being left empty in the database write window.

A grouped write does not have to write all the objects in the Rules Element's working memory to the database. The In field allows an In list of objects and/or classes to be specified which will be written to the database. The class specifications are actually existential pattern matching operations, which allows even finer filtering of the objects if desired.

To even further limit which records are updated or written, a WHERE clause may be included in the Query field to select which records will be updated based on their contents.

Finally, which objects are ultimately written can be controlled by whether or not a record already exists to represent it. If the record doesn't exist, a record can be created to hold it.

Fields

To build a grouped write, complete the Write screen in the Database Editor window as follows.

- Specify Write as the operator
- As the first operand of the Write, specify the database access string if a relational database is being accessed. If a flat file database is being accessed, specify the file name. See the Access String Specification topic for more information.
- In the database Write window, click on the appropriate selection in the Database Type field for the database being written to.
- The Begin field should contain whatever is appropriate for your database. See the Beginning Database Operations topic for more information.
- For a relational database, specify the table name to be accessed in the Query field. If you want to limit the records updated by the write, you can also include a SQL query (for relational databases) or a Rules Element SQL-like query (for flat file databases) in this field. See the Query Write Operations topic for more information on filling in the Query field.

- The End field should contain whatever is appropriate for your database to end a transaction. For almost all relational databases, either "COMMIT" or "COMMIT RELEASE" should be specified. See the Ending Database Operations topic for more information.
- The Name field is used to construct record "keys" by which the objects will be correlated with records in the database. The keys are built dynamically using the object name. See Writing by Key under the Record Specification for Writes topic for more information.
- The Cursor field MUST be left empty
- The In field is used to specify a list of objects and/or classes which will be written to the database. See the Slot Specification for Writes topic for more information.
- If records are to be added and it is not known whether a corresponding record exists to hold an object, then the Create New Record box should be checked.
- If records are to be added and it is known in advance that no corresponding record exists to hold an object, then the Insert Only box should be checked.
- In the Rules Element Properties column, specify the property slots which are to be written to the fields in the database. In the database fields column, specify the corresponding field which is to receive each property slot. See the Slot Specification for Writes topic for more information.

Related Topics

Writing to Databases	In Filtering List
Name	Field Name Specification
Slot Specification for Writes	Record Specification for Writes
Beginning Database Operations	Ending Database Operations
Create New Record	Query Write Operations
Insert Only	

Also, look up individual arguments and your database type for more detailed information.

If Change Retrieves

Usage

A retrieve is mostly useful in "if change" actions as a side affect. For example, a slot's change of value could be a "hint" that other data will be needed, and a retrieve in its If Change actions could be use to retrieve that data. Of course, this is a rather indirect approach - it may be more appropriate to include the retrieve in the RHS of a rule or an order of sources.

Remember that all statements in an if change action are ALWAYS executed, so no matter what the results of the retrieve, execution of the If Change will continue with the next statement.

When the Rules Element begins a retrieve operation, it gets the database access string from the first argument of the retrieve statement.

Related Topics

Arguments Overview	Retrieve Operator
Access String	Atomic Retrieve
Sequential Retrieve	Group Retrieve
Left-Hand Side Retrieves	Right-Hand Side Retrieves
Order of Sources Retrieves	Retrieving from Databases

If Change Writes

Usage

Using a Write in a slot's If Change action is very interesting, since it allows an application to immediately reflect a slot's change of value in an external database. This can include the original database that the slot's value was retrieved from, thus changes to a data item can be instantly reflected in the original data source. If the database is shared among multiple users, the change would be reflected to all users when the Rules Element updated the slot's value.

In the car inventory example, a Write could be included in the if change actions for the car's "price" property. If, during the course of the inferencing, the price of a car changed, the write in the if change action would update that car's price in its inventory record. Any subsequent retrieves from the file or database would reflect the new car's price.

This technique has applications anywhere multiple users share data. It has the capability of allowing multiple users to share the results of the Rules Element's inferencing actions since changes to all data - including hypotheses - can be reflected in an external database.

Again, ALL statements in an if change action are always executed, so no matter what the result of the Write, the if change actions will continue executing.

When the Rules Element begins a write operation, it gets the database access string from the first argument of the write statement.

Related Topics

Arguments Overview	Write Operator
Access String	Atomic Write
Sequential Write	Group Write
Left-Hand Side Writes	Right-Hand Side Writes
Order of Sources Writes	Writing to Databases

In List - (@ATOMS)

Usage

The In argument can be specified in grouped retrieve and write operations. It specifies the list of objects or slots to be processed by the transaction. Interpretations and pattern matching constructs can be included in the In list. The items in the list must be separated by commas. In text knowledge bases the formal syntax of the In list is:

```
@ATOMS=list of generic_atoms;
```

Examples:

- `valve1`; only the object `valve1` will be processed.
- `valve1.state`; only the object `valve1` will be processed and only its property slot `state` will be retrieved or written.
- `\theTank\.fluid`; only one slot will be processed. The string `\theTank\` will be interpreted to yield the object to be processed.
- `<sensors>`; all the objects in the list `<sensors>` will be processed. This pattern matching list will be a subset of the objects in the class `sensors` if it results from the evaluation of one or more conditions in the rule in which the Retrieve or Write statement appears.
- `valve1, \theTank\.fluid, <sensors>`; all the objects and slots previously described will be processed.

Specification

For grouped write operations, the Rules Element takes a group of objects and writes the same property slots from each object to the database. The group of objects is specified in the In list, which can contain lists of object names or class specifications. The properties are specified in the Fields and Properties list in the write window.

Both object and class names may be used in the same operation.

When object names are specified, they are passed to the Rules Element directly. For example, if the In list contained "`car_1, car_2, car_3`", then the objects `car_1`, `car_2` and `car_3` are passed to the Rules Element.

A class name is passed to the Rules Element database interface by enclosing it in angle brackets. For example, to pass all the objects in `car_class` to the database interface, `<car_class>` should be specified in the In list.

Actually, specifying `<class_name>` is an existential pattern matching operation with no "test", therefore all objects currently in the pattern matching list will be passed to the database interface. If this is the first time the particular `<class_name>` specification appears in the LHS or RHS of the rule, then all the objects in the class will be passed. However, if previous pattern matching operations had "trimmed" the list, then only those objects remaining in the list will be passed.

For example, assume that the class `car_class` contains three objects - `car_1`, `car_2`, and `car_3`, and these objects have the string property `Sportive`. Also, assume that only `car_2`'s `Sportive` property contains a value of `Yes`.

In a rule where the LHS contains only a Write operation with an In list of `<car_class>`, ALL of the objects will be passed to the Rules Element, and `car_1`, `car_2`, and `car_3` will be written.

If the LHS has a statement like the following preceding the write:

```
= <car_class>.Sportive"True"
```

Then ONLY `car_2` will be passed to the database interface. This is because the pattern match will have trimmed the list `<car_class>` to only those objects with a `Sportive` property of `Yes`.

This capability is much like being able to do a query across the objects in the Rules Element's working memory and passing only those objects which meet the query criteria to the Rules Element.

How to Specify a List of Object or Class Names

The objects or classes (or, more precisely, the existential pattern matching lists) are passed to the Rules Element in the In field of the database Write window. The following example shows how the class `<car_class>` would be passed to the database interface in a grouped write operation:

The screenshot shows the 'Database Write' dialog box. At the top is a large text area. Below it are several input fields: 'Begin', 'Query' (containing 'CARS'), 'End', 'Name' (containing 'lname1'), 'In' (containing '<cars_class>'), 'Cursor', and 'SqlError'. To the right of these fields is a 'Database Type' dropdown menu set to 'Oracle 7 Driver'. Below the input fields is a section with two tabs: 'Database Fields' and 'Object Properties'. The 'Database Fields' tab is active, showing a table with three rows: 'DB_MODEL' with value 'Model', 'DB_MODEL_DATE' with value 'Model_Date', and 'DB_PRICE' with value 'Price'. To the right of this table are four checkboxes: 'Create New Record' (checked), 'Write Unknown' (unchecked), 'New File' (unchecked), and 'Insert Only' (unchecked). At the bottom right are 'OK' and 'Cancel' buttons.

Figure 3-6 Writing All the Objects in "car_class"

Related Topics

Database Editor Windows
Arguments Overview
Sequential Write
Sequential Retrieve
Dynamic Values

Existence Filtering Operations
Grouped Write
Grouped Retrieve
Interpretations @V(...)

INFORMIX

The Rules Element INFORMIX database interface is only available on certain Unix platforms, and is not currently available under other operating systems (i.e. Mac, PC, Mainframe, VAX/VMS).

INFORMIX-Online is the relational database product of Informix Software, Inc. The query language of INFORMIX is the standard SQL (Structured Query Language) language. This section assumes familiarity with the SQL language and the INFORMIX product.

The Rules Element INFORMIX interface is available as a separate package. An installation guide is provided with the software. It contains all the information required to configure the system and install the database interface.

The basic logic controlling the transactions has been described in the Retrieve and Write topics in this chapter. This part will explain how the SQL queries are constructed.

Database Access String

As explained in the Access String topic in this chapter, the first argument of the Retrieve or Write operators contains the information required to establish the connection with the database. In order to connect with the Informix database server, you must specify the database name and optionally the name of the server on which you wish to use it. The syntax takes the form:

```
"DatabaseName@servername"
```

For more information, please consult your database administrator or refer to the section "Database Name" in the INFORMIX Guide to SQL Reference December 1991 included in your INFORMIX 5 distribution.

On the PC several additional connection parameters are optional.

```
"DatabaseName@servername username password host service  
protocol"
```

For example,

```
"customerdb@hyperion scott tiger jupiter sqlexec tcp-ip"
```

Each parameter must be delimited by a blank space.

Note: Entering the username, password, host, service, and protocol parameters in your connection string may have no effect on establishing the connection. Consult your database administrator to determine whether your database configuration uses these optional parameters.

You cannot be connected to several databases simultaneously. You can nevertheless close a connection by issuing a RELEASE statement (see End string description below) and open a connection to another database afterwards.

Query Syntax

Begin and End Strings

In these strings, you can specify any valid SQL statement which will be sent to the DBMS server. If you want to send several SQL statements, you must separate them by a semi-colon character (;).

The Rules Element recognizes the special words COMMIT, ROLLBACK, and RELEASE in the End statement because they need to be processed differently by the INFORMIX connection module. If COMMIT is encountered, the Rules Element commits the current transaction. If ROLLBACK is encountered the transaction is rolled back and if RELEASE is found, the Rules Element closes the connection with the database.

Usually, in the case of a Write transaction, the Begin statement contains a BEGIN WORK and the End statement contains a COMMIT WORK. A COMMIT will generally be translated to COMMIT WORK. Note that a RELEASE or ROLLBACK assumes a BEGIN WORK has been done. If this is not the case, INFORMIX will generate a warning. You should not be concerned if you see this. You are most likely to encounter this warning after selecting the Restart Session option (which does a ROLLBACK).

Query String

The query string contains one or several table names followed by an optional where clause.

Let us take an example. Our database contains two tables:

- employees with the fields emp_id, name, dept_id, salary and bonus.
- departments with the fields dept_id, name, budget.

You can retrieve all the employee records with the following query:

```
@QUERY= "employees";
```

Note: In the Database Editor, you should not enclose your string in double quotes. You should type only the word “employees.”

You can express complex queries such as:

- (a) @QUERY= "employees where salary > 3000";
 (b) @QUERY= "employees, departments where salary > 3000 and employee.dept_id = department.dept_id";

In the second case (b), the query will join the two tables employees and departments.

The query string is not sent as is to the DBMS server (it is not a valid SQL statement). The actual SQL query is built in the following way:

- If a Name is specified (grouped queries), the Rules Element extracts the field1 and the optional field2...field5 information from the Name.
- Then the Rules Element builds the SELECT statement:

```
SELECT field1, field2,...field5, list_of_fields FROM
query_string
```

where list_of_fields is the list of fields specified in the left part of the double list box of the Database Editor (@FIELDS).

The resulting string would be the string used with the INFORMIX isql utility. isql displays the results of the query on the terminal but the Rules

Element needs to assign the retrieved values to some internal variables. Let us consider our example query string (b). If the name slot of our Database Editor contains 'emp_!emp_id!', and the fields list contains the three properties name, employees.dept_id and salary, then the following string will be sent to the INFORMIX server:

```
SELECT emp_id, name, employees.dept_id, salary FROM employees, departments
WHERE salary > 3000 and employee.dept_id = department.dept_id
```

You must fully specify field names which are present in more than one relation. In our example, dept_id must be prefixed by a table name (even if the two tables contain the same value for this field as a result of our join operation).

You can use the full power of the SQL language and specify expressions instead of field names (i.e. write salary + bonus instead of salary) as long as the SQL string which will be generated is a valid SELECT statement.

Writing Parameterized Queries

You can use either the @V(obj.prop) syntax or the query argument box to parameterize your queries. If you use the query argument box, then you should specify the parameters to be supplied to INFORMIX as "?". The previous example can be transformed as follows:

```
@QUERY= "employees, departments where salary > @V(@SELF.amount) and
employee.dept_id = department.dept_id";
```

or

```
@QUERY= "employees, departments where salary > ? and employee.dept_id =
department.dept_id"; @ARGS= SELF.amount;
```

Note: SELF and interpretations are allowed in the right part of the fields/properties list box (@SLOTS) in the case of sequential or atomic queries (grouped queries use a list of properties, not slots). SELF is allowed only if the query is placed in methods.

Update and Insert Statements

UPDATE and INSERT statements are constructed in a similar way. INSERT statements are generated only if the Create New Record option is selected and will concern only the objects specified in the In list which do not already have a matching record in the database.

The UPDATE statement is generated as follows:

```
UPDATE tables_from_query_string SET list_of_fields/values WHERE
[field1 = value_of_field1 [AND field2 = value_of_field2]...] [AND]
[where_clause_from_query_string]
```

The square brackets indicate optional strings. The field values are passed in a special descriptor area, but their places are identified with "?". Let us take our example (a) and suppose that the salary field needs to be updated and that the Name cell contains 'emp!emp_id!'. The resulting SQL statement will be:

```
UPDATE employees SET salary = ? WHERE emp_id = ? and salary >
3000
```

Note: In that example, the last part of the statement (and salary > 3000) is probably useless.

The INSERT statement is built from the following model:

```
INSERT INTO table_from_query_string ([field1, ][field2, ...] list_of_fields)
VALUES ([value_of_field1, ][value_of_field2, ] ...)
```

Our update example becomes:

```
INSERT INTO employees (emp_id, salary) VALUES (?, ?)
```

The INSERT statement is limited to the first table specified in the query string. You can insert records only into real tables, not into views.

Sequential Queries

In the current implementation, you cannot have more than three active queries simultaneously. You are limited to three active sequential queries or one grouped or atomic query when two sequential queries are pending.

Sequential Write operations are not implemented. You can easily replace a sequential write by an atomic write.

Error Reporting

The Rules Element will report any SQL error message generated by INFORMIX in the transcript window (if this window is write enabled). It will also generate error messages if it encounters problems while building the SQL strings. You can consult the various INFORMIX manuals for a detailed explanation of the messages.

Retrieve Datatype Mapping

The following table indicates how various INFORMIX datatypes may (or may not) be retrieved into various Rules Element datatypes. The Rules Element datatypes are listed (underlined> across the top; the INFORMIX datatypes are listed in the column to the left. A "Y" means that the operation works with no additional effort or concerns. A number means that the operation is possible, but you should see the notes that appear below the table for additional details. A "--" means that the operation is not possible.

	<u>Integer</u>	<u>Float</u>	<u>Boolean</u>	<u>String</u>	<u>Date</u>
integer	Y	1	5	Y	--
smallint	Y	1	5	Y	--
float	--	Y	5	Y	--
real	--	Y	5	Y	--
char(n)	2	2	5	Y	4
varchar(n)	2	2	5	Y	4
date	--	--	5	Y	3
long	--	--	--	--	--
rowid	--	--	--	--	--
raw(n)	--	--	--	--	--

The following notes correspond to the table shown above.

Notes:

1. Conversion from integer to float will automatically take place.
2. If the string contains the proper numeric type requested, it will be copied into the Rules Element property. Otherwise, formats will be required.
3. Informix requires a special Rules Element format be defined in order to retrieve this into a date property. Since the standard INFORMIX date format is "mm/dd/yyyy", a Rules Element format that will accept this format is 'm"/"d"/"yyyy'. This method makes the Rules Element conform to the INFORMIX time format (note that information concerning hours / minutes / seconds is not available).
4. If the string contains a valid date, the Rules Element will take it if provided in the default Rules Element date format ('Mmm dd yyyy hh:mm:ss;mm dd yy hh:mm:ss;Mmm dd yyyy;mm dd yy;'). If in some other format, a format may be attached to the property to allow its acceptance (e.g. a format of 'mm"/"dd"/"yy' would accept "12/25/90").
5. Formats may be applied to treat most datatypes as booleans, though the most obvious / preferred datatypes for this purpose are strings and integers. A default property has been defined so that any string of the form "True" or "False" (case-insensitive) will be converted to the appropriate Rules Element boolean. For example, if you have integers that are "0" for "False" and "1" for "True", you could assign a format of "True;False;1;0;" (which make it print out as True/False, even though it comes in as 1/0). In another example, a Rules Element boolean could be used to indicate all people born in 1990 by reading date fields from the database using the format: "True;False;*/"*/"1990;*".
6. The INFORMIX money type returns a dollar sign ("\$\$") that must be accounted for with a Rules Element format statement. A format that will allow loading INFORMIX money into a Rules Element float is "\$"0.0d'. To load INFORMIX money into a Rules Element integer, you should use "\$"d*' (note that this will truncate the decimal/cents portion of the field).
7. It is possible to do a "non-standard" retrieve from the various INFORMIX datatypes into a Rules Element date slot. However it requires use of the Rules Element formats, and typically results in a peculiar mapping from INFORMIX type to the Rules Element type. This mapping, while possible, is not a preferred way to read integer or floating data from the database, or to load a Rules Element date slot. For reference, formats that could be used are 'yyyy' or 'm"."yy', to load from an integer or float field, respectively.
8. This is possible, but is not a preferred way to read a date from INFORMIX or load a Rules Element float or integer, and does result in loss of information. However, you might need to read an INFORMIX date, and put the year directly into a Rules Element float. You could do such a thing with the following format: '*"/"*/"0.0d'. The desired field could equally well have been the month or day. To load an integer with the year, you could use '*"/"*/"d'.
9. In order to load any kind of INFORMIX floating point number into a Rules Element integer, you must specify a format that will result in truncation of the decimal portion of the number. A format that will work is: 'd*'. Note that you do have to worry about overflow, since a

Rules Element integer is a 32 bit signed quantity, and floating point numbers can be larger than this.

Write Datatype Mapping

The following table indicates how various INFORMIX datatypes may (or may not) be written into from various Rules Element datatypes. The Rules Element datatypes are listed (underlined> across the top; the INFORMIX datatypes are listed in the column to the left. A "Y" means that the operation works with no additional effort or concerns. A number means that the operation is possible, but you should see the notes that appear below the table for additional details. A "--" means that the operation is not possible.

<u>Integer</u>	<u>Float</u>	<u>Boolean</u>	<u>String</u>	<u>Date</u>
Y	5	4	3	--
Y	5	4	3	--
Y	Y	4	3	--
Y	Y	4	3	--
Y	Y	Y	Y	Y
Y	Y	Y	Y	Y
--	--	4	2	1
--	--	--	--	--
--	--	--	--	--

The following notes correspond to the table shown above.

Notes:

1. A special Rules Element format must be defined in order to write into an INFORMIX date field. The standard INFORMIX date format is "mm/dd/yyyy". A Rules Element format that will generate this format is 'mm"/"dd"/"yyyy'. This method makes the Rules Element conform to the default INFORMIX time format (this date format does not support the hours / minutes / seconds fields).
2. If the string contains a valid date, INFORMIX will take it if provided in the standard INFORMIX date format (see note 1).
3. If the string contains the proper numeric type requested, it will be copied into the Informix field. See also note 7.
4. Formats must be applied to treat booleans as non-string INFORMIX datatypes. For example, you could write into an integer field if you use a boolean format of '1;0;True;False' (which accepts True/False, though prints out as 1/0). The most obvious candidates to use for storing booleans are string and integer datatypes. (Strings will directly receive True/False with the default Rules Element format).
5. Floats will be truncated, as necessary, when stored in integer fields. See also note 7.
6. This requires the use of special formats, and is not a preferred or recommended way to store values into the specific INFORMIX fields. For example, one would almost never use Rules Element dates to hold INFORMIX integers, or vice versa. It could be done, but might place restrictions on the values that may be stored.

7. Overflow is possible in certain cases if the input field is larger than the database datatype supports (e.g. storing a Rules Element integer into an INFORMIX smallint). It is also possible to lose precision by, for example, storing the Rules Element integers or floats (double precision) into INFORMIX smallfloats (single precision).

Notes

The main differences between INFORMIX and the screen captures documented in Appendix A, "Database Integration Examples" are as follows:

- You must remember to specify INFORMIX in the Database Editor window (or in the TKB, @TYPE=INFORMIX).
- You must specify parameterized queries as "?", rather than ":val".

Related Topics

Databases
Retrieving from Databases
Writing to Databases

INGRES

INGRES is the relational database product of INGRES Corporation. The query language of INGRES is the standard SQL (Structured Query Language) language. This section assumes familiarity with the SQL language and the INGRES product.

The Rules Element INGRES database interface is available as a separate package. An installation guide is provided with the software. It contains all the information required to configure the system and install the database interface.

The basic logic controlling the transactions has been described under the Retrieve and Write topics in this chapter. This part will explain how the SQL queries are constructed.

Database Access String

As explained under the Access String topic, the first argument of the Retrieve or Write operators contains the information required to establish the connection with the database. In order to connect with the INGRES database server, you must specify the virtual node, database name, and the user name:

```
"virtualnode database username options"
```

For example,

```
"sun10 iidbdb scott"
```

Each parameter must be delimited by a blank space. You should consult your database administrator or Ingres manuals for the exact information about the connection parameters.

You cannot be connected to several accounts simultaneously. You can, however, close a connection by issuing a `RELEASE` statement (see End string description below) and open a connection to another account afterwards.

Query Syntax

Begin and End strings

In these strings, you can specify any valid SQL statement which will be sent to the DBMS server. If you want to send several SQL statements, you must separate them by a semi-colon character (`;`).

The Rules Element recognizes the special words `COMMIT`, `ROLLBACK`, and `RELEASE` in the End statement because they need to be processed differently by the INGRES connection module. If `COMMIT` is encountered, the Rules Element commits the current transaction. If `ROLLBACK` is encountered the transaction is rolled back and if `RELEASE` is found, the Rules Element closes the connection with the database via the `DISCONNECT` statement.

Usually, the Begin statement is left empty and the End statement contains a `COMMIT` in the case of a Write transaction. You could alternatively specify `ROLLBACK` if you wish to undo the effects of your current transaction:

```
@END= "commit";
@END= "rollback";
```

By default, the Rules Element does a `ROLLBACK` when a Restart Session is done.

If the Rules Element is able to communicate with the INGRES database server, but INGRES is unable to open the table (typically because it is locked by some other user/application), the Rules Element will wait until access is allowed. It is possible to use special syntax in the `BEGIN` field to cause INGRES to give up after a specified time. The syntax for this is, for the first query, to specify:

```
@BEGIN= "set lockmode session where timeout = n";
```

where "n" is the number of seconds you are willing to wait while trying to establish the connection to the INGRES database.

It is also possible to use the `BEGIN` field to tell INGRES that you wish to automatically do a `COMMIT` following each transaction:

```
@BEGIN= "set autocommit on";
```

Query string

The query string contains one or several table names followed by an optional where clause.

Let us take an example. Our database contains two tables:

- `employees` with the fields `emp_id`, `name`, `dept_id`, `salary` and `bonus`.
- `departments` with the fields `dept_id`, `name`, `budget`.

You can retrieve all the employee records with the following query:

```
@QUERY= "employees";
```

Note: In the Database Editor, you should not enclose your string in double quotes. You should type only the word `employees`.

You can express complex queries such as:

- (a) @QUERY= "employees where salary > 3000";
 (b) @QUERY= "employees, departments where salary > 3000 and employee.dept_id = department.dept_id";

In the second case (b), the query will join the two tables employees and departments.

The query string is not sent as is to the DBMS server (it is not a valid SQL statement). The actual SQL query is built in the following way:

- If a Name is specified (grouped queries), the Rules Element extracts the field1 and the optional field2...field5 information from the Name.
- Then the Rules Element builds the SELECT statement:

```
SELECT field1, field2,...,field5, list_of_fields FROM
query_string
```

where list_of_fields is the list of fields specified in the left part of the double list box of the Database Editor (@FIELDS).

The resulting string would be the string used with the SQL utility. SQL displays the results of the query on the terminal but the Rules Element needs to assign the retrieved values to some internal variables. Let us consider our example query string (b). If the name slot of our Database Editor contains 'emp_'emp_id!, and the fields list contains the three properties name, employees.dept_id and salary, then the following string will be sent to the INGRES server:

```
SELECT emp_id, name, employees.dept_id, salary FROM employees,
departments WHERE salary > 3000 and employee.dept_id =
department.dept_id
```

You must fully specify field names which are present in more than one relation. In our example, dept_id must be prefixed by a table name (even if the two tables contain the same value for this field as a result of our join operation).

You can use the full power of the SQL language and specify expressions instead of field names (i.e. write salary + bonus instead of salary) as long as the SQL string which will be generated is a valid SELECT statement. The INGRES SQL Reference Manual provides detailed information on SQL.

Writing parameterized queries

You can use either the @V(obj.prop) special syntax or the query argument box to parameterize your queries. Our previous example can be transformed as follows:

```
@QUERY= "employees, departments where salary > @V(@SELF.amount) and
employee.dept_id = department.dept_id";
```

or

```
@QUERY= "employees, departments where salary > :v1 and employee.dept_id =
department.dept_id";@ARGS= SELF.amount;
```

Note: SELF and interpretations are allowed in the right part of the fields/properties list box (@SLOTS) in the case of sequential or atomic queries (grouped queries use a list of properties, not slots). SELF is allowed only if the query is placed in methods.

Update and Insert statements

UPDATE and INSERT statements are constructed in a similar way. INSERT statements are generated only if the Create New Record option is selected and will concern only the objects specified in the In list which do not already have a matching record in the database.

The UPDATE statement is generated as follows:

```
UPDATE tables_from_query_string SET list_of_fields/values
WHERE [field1 = value_of_field1 [AND field2 =
value_of_field2]...] [AND] [where_clause_from_query_string]
```

The square brackets indicate optional strings. Let us take our example (a) and suppose that the salary field needs to be updated and that the Name cell contains 'emp'!emp_id!. The resulting SQL statement will be:

```
UPDATE employees SET salary = 5000 WHERE emp_id = '104' and
salary > 3000
```

Note: In this example, the new salary information and the emp_id is obtained from the object identified by the Name field (e.g. 'emp104'). Also, the last part of the statement (and salary > 3000) is probably useless.

The INSERT statement is built from the following model:

```
INSERT INTO table_from_query_string ([field1, ][field2, ...] list_of_fields)
VALUES ([val1, ][val2, ] ...)
```

Our update example becomes:

```
INSERT INTO employees (emp_id, salary) VALUES ('105', 6500)
```

The INSERT statement is limited to the first table specified in the query string. You can insert records only into real tables, not into views.

Sequential queries

In the current implementation, you cannot have more than three active queries simultaneously. You are limited to three active sequential queries or one grouped or atomic query when two sequential queries are pending.

Sequential writes are not implemented. You can easily replace a sequential write by an atomic write.

Error Reporting

The Rules Element will report any SQL error message generated by INGRES in the transcript window (if this window is write enabled). It will also generate error messages if it encounters problems while building the SQL strings. You can also consult the appropriate INGRES manuals for a detailed explanation of the INGRES messages.

Retrieve Datatype Mapping

The following table indicates how various INGRES datatypes may (or may not) be retrieved into various Rules Element datatypes. The Rules Element datatypes are listed (underlined> across the top; the INGRES datatypes are listed in the column to the left. A "Y" means that the operation works with no additional effort or concerns. A number means that the operation is

possible, but you should see the notes that appear below the table for additional details. A "--" means that the operation is not possible.

Integer	Float	Boolean	String	Date
Y	5	4	3	--
Y	5	4	3	--
Y	Y	4	3	--
Y	Y	4	3	--
Y	Y	Y	Y	Y
Y	Y	Y	Y	Y
--	--	4	2	1
--	--	--	--	--
--	--	--	--	--

The following notes correspond to the table shown above.

Notes

1. Conversion from an integer value to a float will take place.
2. If the string contains the requested numeric type, it will be copied into the Rules Element property.
3. A special Rules Element format must be defined in order to retrieve this field into a date property. A format that should work is 'd'-'mmm'-'yyyy' "h":'mm':"ss'.
4. If the string contains a valid date, the Rules Element will take it if provided in the default Rules Element date format ('Mmm dd yyyy hh:mm:ss;mm dd yy hh:mm:ss;Mmm dd yyyy;mm dd yy;'). If in some other format, a format may be attached to the property to allow its acceptance (e.g. a format of 'mm'/'dd'/'yy' would accept a string containing "12/25/90").
5. Formats may be applied to treat most datatype as booleans. By default, the Rules Element will convert any string of the form "True" or "False" (case-insensitive) to the appropriate Rules Element boolean. The most obvious field types to read into booleans are the various strings and integers. For example, if you have integers that are "0" for "False" and "1" for "True", you could assign a format of '!True;!False;1;0;' (which makes the Rules Element print it out as True/False, even though it comes in as 1/0).
6. A special the Rules Element format is needed to accept this, which ends up discarding the floating point portion (there will be problems if an exponent is present). For example, you could use the following format: 'd*;',.

Write Datatype Mapping

The following table indicates how various INGRES datatypes may (or may not) be written into from various Rules Element datatypes. The Rules Element datatypes are listed (underlined> across the top; the INGRES datatypes are listed in the column to the left. A "Y" means that the operation works with no additional effort or concerns. A number means that the

operation is possible, but you should see the notes that appear below the table for additional details. A "--" means that the operation is not possible.

Note that the Rules Element INGRES database interface needs to make extensive use of the "{I}" syntax for integer database field names in the Name field and the Fields list. This instructs the database interface to not treat this as a string, but rather as a numeric field (e.g. integer).

	Integer	Float	Boolean	String	Date
int	Y	1	1, 4, 5	1, 5	1, 4, 5
smallint	1	1	1, 4, 5	1, 5	1, 4, 5
tinyint	1	1	1, 4, 5	1, 5	1, 4, 5
float	1	Y	1, 4, 5	1, 5	1, 4, 5
char(n)	--	--	Y	Y	Y
varchar(n)	--	--	Y	Y	Y
bit	3	3	1, 4, 5	1, 5	1, 4, 5
money	2	--	--	1, 5	1, 4, 5
date	--	--	4	1, 5	Y
text	--	--	--	--	--
binary(n)	--	--	--	--	--
varbinary(n)	--	--	--	--	--
image	--	--	--	--	--
timestamp	--	--	--	--	--

The following notes correspond to the table shown above.

Notes

1. Datatype conversion, as appropriate and if possible, will take place. For example, a Rules Element integer can be placed into an INGRES integer1 (8 bits), but it must have a value in the allowed range. If the number overflows the fieldwidth, INGRES will not always generate an error, and the value written is not always predictable.
2. There are no "cents" passed in. The integer is treated as an integer number of dollars ("\$").
3. Formats must be applied to treat booleans or dates as various INGRES datatypes. For example, you could write a boolean into an integer field if you use a boolean format of '1;0;True;False' (which accepts True/False, though prints out as 1/0). The most obvious candidates to use for storing booleans are the various string and integer formats. (Strings will directly receive True/False with the default Rules Element format).
4. Since this INGRES field needs to be entered without quotes, but the Rules Element, by default, will put quotes around the field values, the "{I}" prefix syntax must be used for the database field name to indicate that this is a numeric-like field and the Rules Element should not provide quotes.

5. You must be sure to specify a date field that INGRES will accept. Otherwise, with certain platforms and INGRES versions, the database server has been known to crash. A format that is acceptable is: 'd'-'mmm'-'yyyy' 'h':'mm':'ss'.
6. Typically not used in this manner, but possible if the integer contains, for example, "mmddy" (a valid INGRES date input format).

Notes

The main difference between INGRES and the screen captures documented in Appendix A, "Database Integration Examples" are as follows:

1. You must remember to specify INGRES in the Database Editor window (or in the TKB, @TYPE=INGRES).
2. When writing numeric fields, you must use the "{I}" syntax to let the database interface know that it must not provide quotes around the database field being sent from the Rules Element. For example (e.g. ex02ing.tkb):
3. @FIELDS= "{I}DB_PRICE", "DB_MODEL_DATE", "DB_SPORTIVE";
4. In all of the examples where you are going to retrieve from a table, the INGRES interface is generally exactly the same as the standard examples.

Related Topics

Databases
Retrieving from Databases
Writing to Databases

Insert Only - (@FILL)

Usage

Insert Only specifies that a new record be created automatically without first performing an update to existing records. This can be useful when you know in advance that none of the records being written from the Rules Element currently exist in the database. Duplicate records may result if an insert is performed and the record already exists. However, using the Insert Only setting instead of the Create New Record setting produces a significant performance boost since there is no update to perform before inserting the new records.

In the write dialog screen this setting can be specified by clicking in the Insert Only check box. In a text format knowledge base it will appear as:

```
@FILL=INSERT;
```

When Insert Only is selected, do not select Create New Record or New File since these settings are mutually exclusive.

Related Topics

Grouped Write	Arguments Overview
Database Editor Windows	Create New Record
Writing to Databases	New File

Interpretations - @V(...)

Usage

The Rules Element allows you to use the syntax @V(obj.prop) (or equivalently @V(slot)) for more flexibility in parameterizing your knowledge base. This syntax also proves to be very useful with the Rules Element database interface.

As a reminder, you can parameterize your query using a :val syntax (!val for RDB) and specifying a Query Arguments list as in:

```
@QUERY= 'CARS WHERE MODEL = :val1 AND PRICE < :val2'; @ARGS=
car.model,car.price ;
```

where the values found in the Rules Element slots `car.model` and `car.price` will be used to select the appropriate record from the database. For example, `car.model` could be a string slot containing a model name like `FORD`, and `car.price` could be an integer slot containing a price like `12500`. There is an implicit issue with quotes in the resulting query statement generated and sent to the database sever. Some query implementations are indifferent to quotes, while others want quotes only in selected areas. Where quotes matter, the Rules Element will typically provide (or not provide) quotes based on the Rules Element property type (not the database type). For example, with RDB, integer and float values are not quoted, but everything else is.

It is also possible for you to use @V to parameterize this query, as in:

```
@QUERY= 'CARS WHERE MODEL="@V(car.model)" AND PRICE <
@V(car.price)';
```

With @V you do not provide the slots in the @ARGS keyword area. It is important to note that the Rules Element does not provide the quotes around the @V that will be required by most databases. Therefore, you should remember to provide the quotes when dealing with database fields like strings, but typically leave them off when dealing with numeric fields.

The choice of one method or the other is largely based on personal preference. Using @V allows you to generate a query that looks more like the normal query that would be generated (e.g. from an interactive SQL interface), and you do not have to remember about @ARGS and :val. In addition, @V gives you control over where quotes are provided and where they are not. The drawback to @V is that the slot referenced is not "compiled", so if an invalid slot is provided, it is not detected until you actually run the application.

Another interesting way to use the @V syntax is as the 1st argument to the Retrieve or Write: the database access string. In this case, your rule would look something like:

```
RETRIEVE      "@V(SLOT)"      [second_argument(s)]
```

There are two advantages to using @V here. The main one is that a password is frequently involved in providing access to a database. Using @V means that this information does not have to be hard-coded in the knowledge base itself (which could raise security issues). The password / access string would still have to be provided by the slot, but it could be filled by doing something more acceptable (e.g. prompting the user). The other advantage is that this mechanism would allow you to totally switch your

database access strings to make a more portable application. You could provide an ORACLE string on one system, and a SYBASE string on another. Unfortunately the entire query cannot be totally parameterized. For example, the @TYPE=database_type field must be fully specified in the knowledge base.

Note that @V can also be used in the BEGIN and END statements in the Retrieve or Write operation, with many of the same advantages listed above. For example, you could have an END statement like:

```
@END= "@V(commit)";
```

where you could have the slot commit contain "commit" for most databases, but "commit transaction" for SYBASE. A similar technique could be applied to the BEGIN statement. The BEGIN statement can provide a lot more generic database access (e.g. creating/dropping tables, deleting records, etc). See the Begin topic for details.

Related Topics

Dynamic Values	Filename Retrieves @F(...)
Beginning Database Operations	Begin
Ending Database Operations	End
Retrieve Operation	Write Operator
Access String Specification	

Left-Hand Side Retrieves

Usage

In the left hand side (LHS) of a rule or method, a retrieve statement is used to fetch data (or facts) relevant to the current rule or chain of reasoning being followed. For example, if the Rules Element is evaluating a set of rules for determining the evaluation of a car dealer's inventory, a retrieve could be used in the LHS of a rule to get all of the car inventory records.

Remember that a retrieve will still return "True" even if no records are fetched. A retrieve ONLY returns "False" when an error occurs.

Depending on the type of retrieve, different strategies can be used to determine if any records were retrieved. For sequential and atomic retrieves, the cursor will be set to a negative value when no records are returned.

For grouped retrieves, there is no direct way to tell how many records were retrieved. If the records were retrieved into a previously empty class, the Length function can be used to determine how many objects are in the class after the retrieve.

When the Rules Element begins a retrieve operation, it gets the database access string from the first argument of the write statement.

Related Topics

Arguments Overview	Retrieve Operator
Access String	Atomic Retrieve

Sequential Retrieve	Group Retrieve
Right-Hand Side Retrieves	If Change Retrieves
Order of Sources Retrieves	Retrieving from Databases

Left-Hand Side Writes

Usage

Write operations are used less often on the left hand side of a rule or method, largely because a Write isn't an action normally taken when testing for a condition or hypothesis.

Like a retrieve, a write only returns "False" if the write fails. You cannot, for example, test to see if a write added or updated any records by testing to see if the write returned "true" or "false". Since a write doesn't affect the objects which are written, it's not possible to use indirect means to see which objects were written, and which weren't.

When the Rules Element begins a write operation, it gets the database access string from the first argument of the retrieve statement.

Related Topics

Arguments Overview	Write Operator
Access String	Atomic Write
Sequential Write	Group Write
Right-Hand Side Writes	If Change Writes
Order of Sources Writes	Writing to Databases

Link To - (@CREATE)

Usage

The Link To argument is only used in the context of grouped retrieves. It specifies a list of classes or objects to which the objects dynamically created by the retrieve will be linked. Interpretations and pattern matching constructs can be included in the Link To list. The items in the list must be separated by commas. The formal syntax of the Link To list is:

```
@CREATE=list of generic_classes or generic_objects;
```

Example:

```
@CREATE=|sensors|,new_object;
```

The objects dynamically created by the retrieve will be linked to the class `sensors` and as sub-objects to the object `new_object`.

Related Topics

- Database Editor Windows
- Grouped Retrieves
- Arguments Overview

Name - (@NAME)

Usage

The Name field is typically used in the context of grouped transactions. It describes the mapping between Rules Element object names and database field names.

During grouped retrieves, the Name field specifies how database field values (!fieldx!) and string constants ('rootx') are to be concatenated to yield names for the dynamic objects created by the query.

During grouped writes, the Name field specifies how Rules Element object names are to be parsed to yield unique database key values for the insert/update database transaction.

Syntax

There are several valid syntactic forms for the Name string:

```
@NAME="!field1!"
@NAME="'root1'!field1!"
@NAME="!field1!'_'!field2!"
@NAME="'root1'!field1!'_'!field2!"
@NAME="'root1'!field1!'root2'!field2!"
```

...and so on up to a maximum of five root/field combinations

When editing the Name field in the retrieve or write dialog screens, do not enclose the entry in double quotes; the Rules Element will insert them automatically. Also, do not exceed the 255 character limit for slot names when specifying the Name string.

For example, if you want to use the second form described above, you type 'root1'!field1!. The root's are string constants and the field's are field names. When processing one record (in Retrieve or Write), the Rules Element will get the values of fieldi as strings. Then it will sequentially go through the various root/field combinations and concatenate the string rooti with the value of fieldi (those which are not specified in the Name string are considered to be empty strings). The result of this concatenation is the name or the object which is associated with the record. Thus the fields are the "keys" which define the mapping between records and objects.

Note: String constants must be delimited by single quotes ('rootx').

Example 1:

```
@NAME="'sensor'!num_id!";
num_id (from database)    object name
  1  sensor1
  2  sensor2
  3  sensor3
```

Example 2:

```
@NAME="'part_'!type!'_'!id!";
type (from database)id (from database)  object name
new           1      part_new1
used          2      part_used2
old           3      part_old3
```


As the Name information is used to associate objects and records, the fields should be chosen so that they provide a unique key in the database (no two records have the same fields combination). Otherwise, there will not be a one to one mapping between objects and records and information may be retrieved from one record, transferred to an object and written back to many records by mistake. Providing additional information in the Query field could reduce some of the ambiguity if the fields do not identify a unique record, but you should be sure you understand the database contents if using this approach.

Related Topics

Grouped Retrieve	Grouped Write
Database Editor Windows	Debugging Operations
Arguments Overview	Object Names In Retrieve Operations
Record Specification for Writes	

Also see the Grouped Retrieve/Write examples in Appendix A, "Database Integration Examples" for further illustrations of the Name field.

New File - (@FILL)

Usage

The New File setting is only meaningful in the context of a grouped write to a flat-file database. New File specifies whether a new spreadsheet file may be created during a grouped write.

In the write dialog screen this setting can be specified by clicking in the New File check box. In a text format knowledge base it will appear as:

```
@FILL=NEW ;
```

When New File is selected, Create New Record is automatically implied. The Insert Only setting is not compatible with either of these settings.

New File cannot be used to automatically create a table in a relational database during a grouped write. Tables must be explicitly created, either in an external application, or in the Begin or End fields in a retrieve or write operation. For flat-file databases, new files will be created according to the format specified in the database type field. These files can then be accessed by other applications like EXCEL, Lotus 1-2-3, or DBase III.

Related Topics

Grouped Write	Arguments Overview
Database Editor Windows	Create New Record
Writing to Databases	Insert Only
Spreadsheets	

NEXPERT Flat-File Formats

These custom Rules Element spreadsheet and database formats offer some advantages:

- **Simplicity and compatibility:** the standard ascii data file can be used on any platform, and simple custom programs can read or write in the same format.
- **Speed:** the read and write access are much faster than with other data files (SYLK, WKS, DBF3).
- **Readability:** the data file can be edited outside the Rules Element with a text editor, or even printed as a report.

They should be used instead of SYLK, WKS, or DBF3 if you do not plan to use your data file outside the Rules Element with an application program (Excel, Lotus 1-2-3, dBaseIII).

NXP File Format

Every slot is stored on a single line. Its name and value are written with the following delimiters:

```
\obj.prop\="value".....
```

or:

```
\obj\="value".....
```

The second form is used to store obj.Value

The 12 dots represent 12 blank characters which are added when the cell is created, so that the same cell can be updated later with a longer value without altering the line length.

The file is terminated by a line of stars (*).

Example of a file with three slots:

```
\problem\="TRUE"
\sensor.pressure\="200.50"
\sensor.location\="blast_furnace"
*****
```

Note: The objects are sorted alphabetically.

The termination of each line is machine dependent: Carriage Return and/or Line Feed.

The Rules Element will not attempt to move data when it replaces a short string value with a longer one. New values will be truncated if they are more than 12 characters longer than the original values. You can use other tools (i.e. sed on UNIX) to extend the lines on an existing NXP file.

The NXP format can be demonstrated with the following rule. The result is more interesting if you add this rule to an existing set of rules (i.e. primer.kb).

```
If      Yes      Write_NXP_file
Then    hypo
And     Write    "test.nxp"      @TYPE=NXP;@FILL=NEW(*)
```

(*) choose NXP in the database list and select the New File button in the Database Editor.

This rule will create a file called `test.nxp` in your current directory. You can open this file with a text editor to see all the slots of the knowledge base (except those which are UNKNOWN) written line by line with their current values.

NXPDB File Format

The records are stored with the following format:

```

      field1|      field2|      field3|      field4|
*****
      val11|      val12|      val13|      val14|
      val21|      val22|      val23|      val24|
      . . .
*****

```

The main characteristics of the NXPDB format are the following:

- It is an ASCII file and thus can be ported from one machine to another (only the End-Of-Line character may differ).
- All the lines have the same length (fixed length record). This length is computed when the file is created by adding the field widths (including separators).
- The first two lines are the file header and describe the fields of the table. The first line contains all the field names separated by vertical bars. It also defines the widths of the fields. The second line is filled with stars (*).
- The last line of stars indicates the end of the file. Any record written after it will be ignored.
- Every line between the second and the last line represents a record. The values are right-aligned in the columns, followed by vertical bars.

The NXPDB format can be demonstrated with the following rule. The result is more interesting if you add this rule to an existing set of rules (eg. `primer.kb`).

```

If      Yes      Write_NXPDB_file
Then    hypo
And     Write    "test.nxp"  @TYPE=NXPDB;@FILL=NEW;(*)

```

(*) choose NXPDB in the database list and select the New File button in the Database Editor.

This rule will create a file `test.nxp` in the current directory. You can open this file to see all the objects and classes of the knowledge base written line by line in records. The two first fields are `Name` and `Value` (30 characters long), followed by the list of properties of the knowledge base. Each object name is written, but only KNOWN values are pasted. The lines may become very long and difficult to read if your knowledge base contains many properties (especially if your text editor wraps lines). This NXPDB file contains a complete dump of the object base.

Specifying Field Widths

NXPDB uses fixed width records and fields. The default field widths depend on the data type of the property:

boolean	Max(10, length of the field name)
integer, float	Max(15, length of the field name)
string, date, time	Max(30, length of the field name)
special property Value	Max(30, length of the field name)

You can override these default values and specify field widths on a property by property basis. The field width information can be edited in the left part of the double list box of the Write Editor (list of fields, @FIELDS keyword). You specify the field width as a number between parentheses after the name of the field. This feature allows you to customize the layout of your NXPDB files so they can be edited easily or printed as reports. You must carefully choose your field widths because the Rules Element will truncate the strings to fit in the space that you have reserved for them. If a string is larger than its field, it will be truncated and some information will be lost. This may be harmless if you want to use the NXPDB file only as a report but problems will arise if the contents of the file are retrieved afterwards.

Examples

<i>List of Fields</i>	<i>List of Properties</i>	<i>Notes</i>
Job(15)	Position	(a)
Salary(10)	Salary	(b)
SS_Number(10)	SS_Number	(b)
Married(5)	MaritalStatus	(c)

(a) The Job field has a maximum width of 15 characters (default is 30 for string, date and time). Your job descriptions must be less than 15 characters wide.

(b) The Salary and SS_Number fields have a maximum width of 10 characters (default is 15 for integer and float). You must take into account the formatting information associated with the property to compute the field width. For example, the Salary property may be formatted as \$ 3000 or 3000 dollars.

(c) The default width for booleans is 10 characters. Five is sufficient for TRUE and FALSE. One character will be enough if your boolean format is "T";"F"; (and if you are also using a 1 letter format for UNKNOWN and NOTKNOWN values).

Notes

The Rules Element will never truncate the field names written in the header of the file. If a field name contains 8 letters and if you specified a field width of 5 characters for it, the Rules Element will use 8 as field width.

Field names are also used in the Name specification (@NAME) which defines the mapping between records and objects ('root1!field1!root2!field2!'). You can also specify a field width for field1 and field2 (i.e.

```
'emp_'!emp_name(12)!).
```

UNKNOWN and NOTKNOWN values are written as UNKNOWN (if Write Unknown is selected) and NOTKNOWN unless you have specified a special format for them (i.e. @N=*; @U=?;). So, your fields should be at

least 8 characters wide if you expect NOTKNOWN or UNKNOWN values and you have not defined a custom format.

Related Topics

Spreadsheets

Database Editor Windows

Arguments Overview

Writing to Databases

Retrieving from Databases

Object Names In Retrieve Operations

Explicit Object Names

In the simplest case, the Retrieve operation explicitly states which slots (object.property combinations) will receive which fields from the database records. This means that no matter what the records or fields contain, the fields will always be mapped to the same slots.

For example, a Retrieve could be coded such that as a car record is retrieved, the fields would be pasted into the slots `MyCar.Name`, `MyCar.Price`, and `MyCar.Model`. These are explicit names: EVERY car's record will be pasted into the `MyCar` object's `Name`, `Price`, and `Model` properties.

Explicit names are used when records are retrieved one by one, as in an atomic or sequential retrieval. With explicit names, a knowledge base will typically retrieve a record, process the slots, and (possibly) go on to retrieve the next record into the same slots.

Explicit names cannot be used with grouped retrieval, since many records are retrieved at once, and each succeeding record's fields would be written over the previous fields in the slots (since only one set of slots can be specified), and all but the last record's fields would be lost.

When a Retrieve operation uses explicit names, it is possible to split a record's fields across several objects by merely specifying slot names (object.property combinations) which are in different objects. For example, a car record's fields could be retrieved into `MyCar.Price`, `YourCar.Model`, and `TheirCar.Model_Date`. However, as discussed before, this is probably only useful in specialized applications since the relationship of the fields is no longer reflected in the Rules Element's object representation.

Constructed Object Names

It's also possible to use data from the record itself to construct the name of the object which will receive the record's fields. All or a portion of the name can be built using the actual data in one or more of the record's fields. If desired, constant strings can be interspersed with the field data when forming the object name.

Take, for example, a car inventory database containing a field `DB_MODEL` and `DB_CAR_NAME` for each car. In this inventory there are four cars whose `DB_MODEL` fields contain `TOYOTA`, `HONDA`, `BMW`, and `MERCEDES`.

These records could be retrieved into four different objects by using the `DB_MODEL` field used to build the name of each object. Thus, the records

could be retrieved into the objects named TOYOTA, HONDA, BMW, and MERCEDES. In this case, the object names are built directly from the database field DB_MODEL. The object name is later combined with the property names to form "object.property" combinations - slot names - to receive the record's field values.

As the Rules Element forms the name for each object, it looks in its working memory for an object with the same name. If the object is found, the Rules Element will update its slots with the fields from the record. If the object is NOT found, the Rules Element can either skip the record, or create a new object for the record.

It is important that the fields and constants used to form the object names result in unique names. If not, the data retrieved into some objects may be lost as later records generate the same object name, and overlay the earlier data. For example, if there were two HONDA records in the car inventory, the data from the second HONDA record retrieved would overlay the first record's data.

To avoid this, include at least one field in the object name whose value will be unique, or combine two or more fields to form a unique value. For example, the previous case could be made unique by using the DB_CAR_NAME field for the object name, or combining the DB_MODEL and DB_CAR_NAME to form the name.

Grouped Retrieve operations MUST use data from the record to construct the object names. A grouped retrieve typically fetches more than one record at once, and the Rules Element must have a way to build multiple object names as the records are retrieved.

Constructing Object Names

You provide the model for constructing the object names in the Name field of the Retrieve window. It is specified as a series of constants (or "roots") and/or field names to be used in constructing the object names. The root fields should be enclosed in single quotes, and the field names in exclamation points ("!"). For example, to specify that the field "DB_MODEL" is to be used as the object name, you would specify "!DB_MODEL!" in the name field.

To combine the "make" field with the constant CAR_, you would specify:

```
' CAR_ ' !DB_MODEL!
```

Even more complex constructs are possible: to combine the DB_MODEL and DB_CAR_NAME field with two roots, you might specify:

```
' CAR_ ' !DB_MODEL! ' _NAME_IS ' !DB_CAR_NAME!
```

It's important to remember that the Name field is composed of field names, not property names. The field names specified must be present in the records being retrieved, otherwise an error will occur.

Field names which occur in the Name field may be repeated in the Fields and Properties list.

Related Topics

Grouped Retrieve	Cursor Slot Specification
Access String	Slot Specification for Retrieves
Query Retrieve Operations	Database Editor Windows
Retrieving from Databases	Name

ORACLE

ORACLE is the relational database product of ORACLE Corporation. The query language of ORACLE is the standard SQL (Structured Query Language) language. This section assumes familiarity with the SQL language and the ORACLE product.

The Rules Element ORACLE database interface is available as a separate package. An installation guide is provided with the software. It contains all the information required to configure the system and install the database interface.

The basic logic controlling the transactions has been described under the Retrieve and Write topics in this chapter. This part will explain how the SQL queries are constructed.

Database Access String

As explained under the Access String topic in this chapter, the first argument of the Retrieve or Write operators contains the information required to establish the connection with the database. In order to connect with the Oracle 7 database server, you must specify the account name and the password.

The syntax is the standard Oracle 7 syntax:

```
"username password host usefixed"
```

For example:

```
"scott tiger t:hyperion:HYPERIONSID"
```

Each parameter must be delimited by a blank space. The host name follows any syntax supported by SQL*Net. Consult your database administrator or Oracle manual for the exact information.

On the PC, Oracle users with SQL*Net 2.0 must provide the full network information; aliases are not supported. On other platforms, aliases are supported.

Note: The "usefixed" parameter controls whether the Oracle 7 CHAR (Type 96) is supported. The default is set to True in order to map fixed and variable length character types as required by Oracle 7. You must set this parameter to False for any connection that you establish to Oracle 6.

You cannot be connected to several accounts simultaneously. You can nevertheless close a connection by issuing a RELEASE statement (see End string description below) and open a connection to another account afterwards.

Query Syntax

Begin and End Strings

In these strings, you can specify any valid SQL statement which will be sent to the DBMS server. If you want to send several SQL statements, you must separate them by a semi-colon character (;).

The Rules Element recognizes the special words COMMIT, ROLLBACK, and RELEASE in the End statement because they need to be processed differently by the ORACLE connection module. If COMMIT is encountered, the Rules Element commits the current transaction. If ROLLBACK is encountered the transaction is rolled back and if RELEASE is found, the Rules Element closes the connection with the database.

Usually, the Begin statement is left empty and the End statement contains a COMMIT in the case of a Write transaction.

Query String

The query string contains one or several table names followed by an optional where clause.

Let us take an example. Our database contains two tables:

- employees with the fields emp_id, name, dept_id, salary and bonus.
- departments with the fields dept_id, name, budget.

You can retrieve all the employee records with the following query:

```
@QUERY= "employees";
```

Note: In the Database Editor, you should not enclose your string in double quotes. You should type only the word employees.

You can express complex queries such as:

- (a) @QUERY= "employees where salary > 3000";
 (b) @QUERY= "employees, departments where salary > 3000 and employee.dept_id = department.dept_id";

In the second case (b), the query will join the two tables employees and departments.

The query string is not sent as is to the DBMS server (it is not a valid SQL statement). The actual SQL query is built in the following way:

- If a Name is specified (grouped queries), the Rules Element extracts the field1 and the optional field2...field5 information from the Name.
- Then the Rules Element builds the SELECT statement:

```
SELECT field1, field2,...field5, list_of_fields FROM
query_string
```

where list_of_fields is the list of fields specified in the left part of the double list box of the Database Editor (@FIELDS).

The resulting string would be the string used with the SQL*Plus utility. SQL*Plus displays the results of the query on the terminal but the Rules Element needs to assign the retrieved values to some internal variables. In fact, the Rules Element inserts an INTO clause before the FROM clause to describe where the values should be returned (see the Pro*C manual for details). Let us consider our example query string (b). If the name slot of our Database Editor contains 'emp_'!emp_id!, and the fields list contains the

three properties name, employees.dept_id and salary, then the following string will be sent to the ORACLE server:

```
SELECT emp_id, name, employees.dept_id, salary INTO :nxp1, :nxp2, :nxp3,
:nxp4 FROM employees, departments WHERE salary > 3000 and employee.dept_id
=department.dept_id
```

The :nxp1, :nxp2 variable syntax is the standard SQL syntax. If you write parameterized queries (see section below), you should choose variable names which do not conflict with these names.

You must fully specify field names which are present in more than one relation. In our example, dept_id must be prefixed by a table name (even if the two tables contain the same value for this field as a result of our join operation).

You can use the full power of the SQL language and specify expressions instead of field names (i.e. write salary + bonus instead of salary) as long as the SQL string which will be generated is a valid SELECT statement. The SQL*Plus User's Guide provides detailed information on SQL.

Writing Parameterized Queries

You can use either the @V(obj.prop) special syntax or the query argument box to parameterize your queries. Our previous example can be transformed as follows:

```
@QUERY= "employees, departments where salary > @V(@SELF.amount) and
employee.dept_id = department.dept_id";
```

or

```
@QUERY= "employees, departments where salary > :v1 and employee.dept_id =
department.dept_id";@ARGS= SELF.amount;
```

Note: SELF and interpretations are allowed in the right part of the fields/properties list box (@SLOTS) in the case of sequential or atomic queries (grouped queries use a list of properties, not slots). SELF is allowed only if the query is placed in methods.

Update and Insert Statements

UPDATE and INSERT statements are constructed in a similar way. INSERT statements are generated only if the Create New Record option is selected and will concern only the objects specified in the In list which do not already have a matching record in the database.

The UPDATE statement is generated as follows:

```
UPDATE tables_from_query_string SET list_of_fields/values WHERE
[field1 = value_of_field1 [AND field2 = value_of_field2]...] [AND]
[where_clause_from_query_string]
```

The square brackets indicate optional strings. Let us take our example (a) and suppose that the salary field needs to be updated and that the Name cell contains 'emp!emp_id!'. The resulting SQL statement will be:

```
UPDATE employees SET salary = :nxp1 WHERE emp_id = :nxp1 and salary > 3000
```

In that example, the last part of the statement (and salary > 3000) is probably useless.

The INSERT statement is built from the following model:

```
INSERT INTO table_from_query_string ([field1, ][field2, ...] list_of_fields)
VALUES (:nxpv1, ][:nxpv2, ] :nxpvi ...)
```

Our update example becomes:

```
INSERT INTO employees (emp_id, salary) VALUES (:nxpv1, :nxpv2)
```

The INSERT statement is limited to the first table specified in the query string. You can insert records only into real tables, not into views.

Sequential Queries

In the current implementation, you cannot have more than three active queries simultaneously. You are limited to three active sequential queries or one grouped or atomic query when two sequential queries are pending.

Sequential Write operations are not implemented. You can easily replace a sequential write by an atomic write.

Error Reporting

The Rules Element will report any SQL error message generated by ORACLE in the transcript window (if this window is write enabled). It will also generate error messages if it encounters problems while building the SQL strings. You can consult the ORACLE Error Messages and Code manual for a detailed explanation of the ORACLE messages.

Retrieve Datatype Mapping

The following table indicates how various ORACLE datatypes may (or may not) be retrieved into various Rules Element datatypes. The Rules Element datatypes are listed (underlined> across the top; the ORACLE datatypes are listed in the column to the left. A "Y" means that the operation works with no additional effort or concerns. A number means that the operation is possible, but you should see the notes that appear below the table for additional details. A "--" means that the operation is not possible.

	<u>Integer</u>	<u>Float</u>	<u>Boolean</u>	<u>String</u>	<u>Date</u>
signed word	1	1	1, 2	1	1, 2
signed longword (scale)	Y	1	1, 2	1	1, 2
signed longword (no scale)	Y	1	1, 2	1	1, 2
signed quadword	Y	1	1, 2	1	1, 2
f_floating	1	1	1, 2	1	1, 2
g_floating	1	1	1, 2	1	1, 2
text	Y	Y	Y	Y	Y
varying string	Y	Y	Y	Y	Y
date	4	--	1, 2	1	3
segmented string	--	--	--	--	--

Notes

1. Conversion, as appropriate and if possible, will take place.
2. If the string contains the proper numeric type requested, it will be copied into the Rules Element property.
3. Oracle requires a special Rules Element format be defined in order to retrieve this into a date property. The "Standard ORACLE DATE" format is "DD-MON-YY" (in Oracle terms). A Rules Element format that will accept this format is 'dd"- "mmm"- "yy'. This method makes the Rules Element conform to the Oracle time format (costing loss of information in the hours/minutes/seconds fields). An alternative is to make Oracle conform to the Rules Element format. To do this requires the user specify an Oracle conversion format. What you should realize is that the retrieve request passes the fieldname listed in the FIELD box on exactly as typed. The database will use this in its retrieve. Since Oracle permits a conversion function in the retrieve, you could have entered: TO_CHAR(date_fieldname,'MON DD YYYY HH24:MI:SS') ... where "date_fieldname" is the name of the date field being retrieved. This will cause the returned date field to be in a form that is directly accepted by the Rules Element (plus it provides the additional time information).
4. If the string contains a valid date, the Rules Element will take it if provided in the default Rules Element date format ('Mmm dd yyyy hh:mm:ss;mm dd yy hh:mm:ss;Mmm dd yyyy;mm dd yy;'). If in some other format, a format may be attached to the property to allow its acceptance (e.g. a format of 'mm"/"dd"/"yy' would accept "12/25/90").
5. Formats may be applied to treat most datatypes as booleans. A default property has been defined so that any string of the form "True" or "False" (case-insensitive) will be converted to the appropriate Rules Element boolean. For example, if you have integers that are "0" for "False" and "1" for "True", you could assign a format of 'True;False;1;0;' (which make it print out as True/False, even though it comes in as 1/0). The most obvious candidates to use for booleans are the various strings and the various integers.

Write Datatype Mapping

The following table indicates how various ORACLE datatypes may (or may not) be written into from various Rules Element datatypes. The Rules Element datatypes are listed (underlined> across the top; the ORACLE datatypes are listed in the column to the left. A "Y" means that the operation works with no additional effort or concerns. A number means that the operation is possible, but you should see the notes that appear below the table for additional details. A "--" means that the operation is not possible.

<u>Integer</u>	<u>Float</u>	<u>Boolean</u>	<u>String</u>	<u>Date</u>
Y	5	4	3	--
Y	5	4	3	--
Y	Y	4	3	--
Y	Y	4	3	--
Y	Y	Y	Y	Y
Y	Y	Y	Y	Y

```
--      --      4      2      1
--      --      --      --      --
--      --      --      --      --
```

Notes

1. Oracle requires a special Rules Element format be defined in order to write into an Oracle date field. The "Standard ORACLE DATE" format is "DD-MON-YY" (in Oracle terms). A Rules Element format that will generate this format is 'dd "-" mmm "-" yy'. This method makes the Rules Element conform to the default Oracle time format (costing loss of information in the hours/minutes/seconds fields).
2. If the string contains a valid date, Oracle will take it if provided in the "Standard ORACLE DATE" format (see note 1).
3. If the string contains the proper numeric type requested, it will be copied into the Oracle field.
4. Formats must be applied to treat booleans as non-string Oracle datatypes. For example, you could write into an integer field if you use a boolean format of '1;0;True;False' (which accepts True/False, though prints out as 1/0). The most obvious candidates to use for storing booleans are the various string and integer formats. (Strings will directly receive True/False with the default Rules Element format).
5. Conversion, as appropriate and if possible, will take place.

Notes

The main difference between ORACLE and the screen captures documented in Appendix A, "Database Integration Examples" are as follows:

1. You must remember to specify ORACLE in the Database Editor window (or in the TKB, @TYPE=ORACLE).
2. There are no differences between the ORACLE examples and the general/generic database examples.

Related Topics

Databases
Retrieving from Databases
Writing to Databases

Order of Sources Retrieves

Usage

An Order of Sources method is an ideal place to use retrieve operations, especially atomic retrieves. This allows you to fetch a slot's value from a database only when it is needed (that is, when a slot is referenced and its value is `UnKnown`).

For example, using the car inventory example again, the `car` object could have a property called `dealer_name` which is NOT included in the "cars" inventory database, and thus remains unknown even if the object's

inventory record is retrieved. Including a retrieve operation in the `dealer_name` method's order of sources will cause the retrieve to be executed **ONLY** if that slot is referenced.

Remember that no matter what the retrieve operation returns, the order of sources will continue execution until a value has been found for the slot. Thus, if the retrieve fails to get a value for the slot - due to an error OR a "no records found" condition - the order of sources will continue execution with the next statement. The statements that follow can pursue alternative sources for the slot's value - including executing additional Retrieve statements.

This behavior can lend itself to very interesting implementations, especially in rich database environments. In the simplest case, multiple Retrieve statements in an order of sources can be used to search a hierarchy of files or databases for a slot's value. This hierarchy could reflect the preferred order of the retrieves since the Rules Element will execute the order of sources top down. Therefore, the first retrieve could be from a table or file with the most preferred data, the second in one with less confidence, and so forth.

An even more interesting approach is possible in distributed database environments - the first retrieve can attempt to access a remote file or database, such as a very large database on a mainframe-type platform. If this fails - due to a communications failure or other problems - subsequent retrieves in the order of sources can access a local, "backup" file or database to satisfy the request. This technique is very useful in applications like credit authorization - which need some data source to complete successfully.

When the Rules Element begins a retrieve operation, it gets the database access string from the first argument of the retrieve statement.

Related Topics

Arguments Overview	Retrieve Operator
Access String	Atomic Retrieve
Sequential Retrieve	Group Retrieve
Left-Hand Side Retrieves	Right-Hand Side Retrieves
If Change Retrieves	Retrieving from Databases

Order of Sources Writes

Usage

The main use for a Write operation in a method's Order of Sources is as a side affect of the Rules Element inquiring as to a slot's value. One possible application of this could be a specialized logging mechanism for making a record when a particular slot is referenced.

Since a write can **NEVER** change a slot's value from `UnKnown`, an order of sources will **ALWAYS** continue execution after a write.

When the Rules Element begins a write operation, it gets the database access string from the first argument of the write statement.

Related Topics

Arguments Overview	Write Operator
Access String	Atomic Write
Sequential Write	Group Write
Left-Hand Side Writes	Right-Hand Side Writes
If Change Writes	Writing to Databases

Properties List - (@PROPS)**Usage**

The properties list can be specified in all types of transactions except operations on spreadsheet files. This list is edited in the right part of the double list box at the bottom of the Database Editor windows.

In the case of a grouped transaction, the list is a list of properties (separated by commas), and it is prefixed by the @PROPS keyword.

```
@PROPS=list of properties;
```

This is very similar to the Slots List (@SLOTS) as described for that topic.

Related Topics

Database Editor Windows	Slots List
Arguments Overview	Retrieving from Databases

Query (@QUERY)**Usage**

The query string contains the record selection statement which will be sent to the Rules Element and/or the DBMS server. The query statements use the query language provided by the underlying database architecture:

- RDO if using RDB
- SQL if using most relational databases
- the Rules Element Query Language if using a flat file

With RDB, the query string is a substring of the start_stream statement which would be written in RDO.

With most relational databases, the query string is a substring of the select statement which would be written in SQL with the appropriate SQL user-interface.

The Rules Element Query Language used with flat files appears under the Query Language topic.

The formal syntax of the query statement is:

```
@QUERY=quoted_string;
```

When you edit the query string with the database editor, you should not enclose it in double quotes. They will be automatically inserted by the Rules Element.

The special constructs @V(obj.prop), @SELF, and @PROP are allowed in the query statement.

If the query string is an interpreted slot (@V(obj.prop)) to yield a formatted date, it must be preceded by the DATE function: DATE(@V(obj.prop)).

Related Topics

Database Editor Windows
Query Retrieve Operations
Query Arguments
Query Write Operations

Arguments Overview
Query Language
Query Example (Sequential Retrieve)

Specific database operations and database topics provide more details and examples on how to use the query statement.

Query Language

This section describes how to use the Rules Element Query Language. You can use the Query Language to query flat-file databases such as Lotus files, NXPDB files, or Excel files. Without the Query Language, you cannot limit the records like you can with relational databases that have their own query languages. The Query Language is based on SQL's Select statement, and you can use it in the Query field of the Retrieve or Write window.

An example of using the Query Language to limit records you retrieve from or write to flat-file databases is given, followed by a description of the structure of the language. This section contains the following topics:

- Example of a query
- Structure of a query
- Values
- Operators: Arithmetic, Relational, Boolean, and Others
- Functions: SUM, MIN, MAX, and Others
- Using Dynamic Values
- Wildcards
- Two Kinds of Errors.

Example of a Query

If you are familiar with other query languages, this query language is a subset because the Rules Element constructs the full query from other information you supply in the Retrieve or Write window. Using SQL terminology, the Rules Element's Query Language consists of the WHERE clause such as shown in this example:

```
select serial_number, price, color from cars where price
between 12000 and 15000 or color like "red"
```

Specify the fields you are selecting, such as serial_number, price, and color, in the Database Fields field.

Instead of specifying a table, such as cars in the above example, specify the filename of the database you are using when you select the Retrieve or Write operators.

Specify the where clause in the Query field of the Retrieve or Write window using the Rules Element's Query Language that is described in this section.

The structure of the Rules Element's Query Language is summarized in the next section.

Structure of a Query

This summarizes how to construct a query using the Rules Element's Query Language:

```
search_criteria
```

```
or
```

```
search_criteria boolean_operator search_criteria
```

```
where search_criteria is:
```

```
expression relational_operator expression
```

```
or
```

```
expression in x : y
```

```
or
```

```
expression between x and y
```

```
or
```

```
expression in [x1, x2, x3]
```

expression is a field name, or field names with arithmetic operators.

boolean_operators and relational_operators are described in the section Operators: Arithmetic, Relational, Boolean, and Others.

x, y, x1, x2, and x3 are values, which are described in the section Values.

Values

Values can be strings, numbers, booleans, dates, or times. Here are examples of each:

Strings	"red" "Miata"
Numbers	15000 1990 19.90
Booleans	True 1 False 0
Dates	DATE(1990, 6, 15) DATE(1990, 12, 25)
Times	TIME(10, 45, 0) TIME(22, 30, 0)

Use values when you are selecting fields from the database based on their value. For example, this query retrieves all records where the price is less than \$15,000 and the car has been sold:

```
price < 15000 and sold = True
```

This example selects all records where the date the car was made is later than December 1, 1989:

```
model_date > DATE(1989, 12, 1)
```

This example selects all records where the time field is less than 1:15 p.m. or the car is in stock:

```
time_stamp < TIME(13,15,0) or in_stock = 1
```

Operators: Arithmetic, Relational, Boolean, and Others

Operators perform an action on values. These are examples of operators:

```
*
/
<
>
=
and
or
contains
```

Operators are arithmetic, relational, boolean, or other. The next four sections describe the types of operators.

Arithmetic Operators: +, -, *, /

Use arithmetic operators to do arithmetic on values. This table lists the arithmetic operators and their descriptions:

Arithmetic Operators	Description
+	Addition
-	Subtraction
*	Multiplication
/	Division

Examples:

This query finds cars that generated a commission of more than \$1,200:

```
(price * (commission_rate/100)) > 1200
```

This query finds cars that, if prices were raised 8 percent, will cost more than \$18,000:

```
(price * 1.08) > 18000
```

Relational Operators: =, <, In, Contains, and Like

Relational operators compare the value of expressions. This table lists the relational operators and their descriptions:

Relational Operators	Description
= == eq	Equal to
!= /= ne	Not equal to
< lt	Less than
<= le	Less than or equal to
> gt	Greater than
>= ge	Greater than or equal to
? like contains	String contains a pattern

Using not:

You can use the modifier not with like, contains, and ? to negate the search query. For example, this query looks for all cars which were not sold in California:

```
city_and_state not contains "California"
```

Examples:

This query finds cars that cost more than 15000:

```
price > 15000
```

This query finds all cars that are not Volkswagens:

```
model != "Volkswagen"
```

These queries finds salespeople whose name contains "John":

```
salesperson contains "John"
salesperson like "John"
salesperson ? "John"
```

Boolean Operators: And's, Or's, and Xor's

These boolean operators take two operands and form an expression that evaluates to true or false. For example, this is an expression that contains the boolean operator and:

```
(price < 15000) and (color = "red")
```

The first operand of and is (price < 15000), and the second operand is (color = "red").

This table lists the boolean operators and their descriptions:

Boolean Operators	Description
and &	Both operands being evaluated must be true for the whole expression to be true.
or	Either operand or both operands being evaluated must be true for the whole expression to be true.
xor #	Either operand must be true but not both for the whole expression to be true (exclusive-or).

Examples:

This query finds cars that satisfy both of these criteria:

- sold by salesperson "Jan"
- cost more than or equal to \$21,000

```
(salesperson like "Jan") and (price ge 21000)
```

This query finds cars that satisfy one of these criteria:

- sold by salesperson "Jan"
- sold by salesperson "Kris"

```
(salesperson ? "Jan") | (salesperson ? "Kris")
```

This query finds all cars that are not Volkswagens or Mazdas:

```
(model = "Volkswagen") xor (model = "Mazda")
```

Other Operators: In, Between

Use these operators to evaluate whether an expression is in a range of values or a list of values. The values can be strings, numbers, dates, or times. For example, this query evaluates whether price is greater than \$10,000 and less than \$15,000:

```
price between 10000 and 15000
```

This table lists the other operators and their descriptions:

Other Operators	Description
<i>value</i> between x and y	Evaluates whether value is greater than x and less than y.
<i>value</i> in x : y	Evaluates whether value is greater than or equal to x and less than or equal to y.
<i>value</i> in [x1, x2, x3]	Evaluates whether value is one of the values listed in brackets.

Using not:

You can use the modifier not with in and between to negate the search query. For example, this query looks for all cars with a price not in the range of \$13,000 and \$18,000:

```
price not between 13000 and 18000
```

This query finds cars that are not Mazdas, Hondas, or Volkswagens:

```
make not in ["Mazda", "Honda", "Volkswagen"]
```

Examples:

This query finds cars that were sold after January 1, 1990 and before June 30, 1990:

```
sold_date between DATE(1990, 1, 1) and DATE(1990, 6, 30)
```

This query finds cars that cost more than \$15,000 and less than \$15,100:

```
price between 15000 : 15100
```

This query finds all cars sold by Alex, Jan, or Kris:

```
salesperson in ["Alex", "Jan", "Kris"]
```

Functions: SUM, MIN, MAX, and Others

You can use functions in your query to a relational database (not supported on other database types). This table lists the functions available in the Rules Element's Query Language and the descriptions of the functions.

Function	Description
AVG(<i>expression</i>)	Compute the average value of all values described by <i>expression</i> .
COUNT(<i>fieldname</i>)	Counts the total number of occurrences of <i>fieldname</i> .
MAX(<i>expression</i>)	Computes the largest value of all the values described by <i>expression</i> .
MIN(<i>expression</i>)	Computes the smallest value of all the values described by <i>expression</i> .
SUM(<i>expression</i>)	Computes the total of all the values described by <i>expression</i> .

Expressions are names of fields, or names of fields with arithmetic operators.

Examples:

This query selects all the cars that cost more than the average price of all the cars:

```
price > AVG(price)
```

Dynamic Values

You can use the current value of the property slot of an object in your query. For example, this query finds the value of MyFavoriteColor.value, blue, and uses it to retrieve all the records that describe a blue car:

```
color contains "@V(MyFavoriteColor.value)"
```

This query finds the value of CurrentCity.value, San Francisco, and uses it to find all records where the car was shipped to San Francisco:

```
shipped_city like "@V(CurrentCity.value)"
```

Warning: For NXPDB, SYLKDB, DBF3, and WKSDB use field names that are in the query, in the Database Fields column, or in the properties list of the Retrieve or Write statement. When the @V contains a character value, it must be enclosed in quotes.

Wildcards

You can use wildcards with strings. Wildcards allow you to specify a pattern to match when doing the query. The Rules Element's Query Language has two wildcards:

?	Replaces one character.
*	Replaces any string.

Examples:

This query finds all records that have an address in California:

```
city_and_state contains "*", California"
```

This query finds all records that have a 4-character serial number that ends in 0:

```
serial_number = "???0"
```

Two Kinds of Errors

When the Rules Element finds an error in the query, such as a misspelling, no records are retrieved. Two errors are:

- Syntax
- Incompatible types

The Rules Element writes error messages to the transcript window. This is an example of a syntax error, because contains is misspelled:

```
city_and_state contains "*", New York"
```

If you try and compare incompatible types, such as numbers and strings, the Rules Element generates an error message. This is an example of incompatible types because the field serial_number is a string:

```
serial_number > 2350
```

Related Topics

Query Retrieve Operations
 Query Arguments
 Query

Query Field in Retrieve Operations

This section discusses how to build the Query field for retrieve operations. The Query allows you to filter incoming records based on the actual data in the record's fields. Two kinds of queries can be used with the Rules Element:

- For relational databases such as Oracle, INGRES, and Sybase, any ANSI-standard SQL query supported by the database may be used. See the appropriate database topic for details.
- For non-relational databases, the Rules Element's own SQL-like query language can be used to filter records. See the Query Language topic for more details.

Query Field

When retrieving records from a relational database such as INGRES, Sybase, Oracle, or SQL/DS, the query is handled by the central database manager or server. Therefore, the query can use whatever implementation of the ANSI SQL standard is supported by the particular database being used.

Keep in mind that using specialized features of a given database will mean that the Retrieve may have to be changed if another database type is used. Generally, if the query uses only those features defined by the ANSI SQL standard, it will be portable across most, if not all, relational database products.

The first thing in the query field must be the table name(s) to be accessed by the retrieve operation. The names can be in any format legal for the database being accessed. This flexibility is important for databases such as SQL/DS which allow you to specify remote table names in a special format. The Rules Element will use the table names "as-is" as it constructs the SQL "SELECT" statement.

If ALL records are to be retrieved, then nothing except the table name should be specified in the Query field.

The second part of the query field is the "WHERE" clause to be included in the SQL "SELECT" statement, and MUST be preceded by the word "WHERE". It is also included "as-is" in the "SELECT" statement constructed by the database interface.

For example, to retrieve only the records from the CARS table in which the DB_SPORTIVE column contains YES, the query field would contain the following:

```
CARS WHERE DB_SPORTIVE = 'YES'
```

More complex queries can be specified, such as:

```
CARS WHERE DB_SPORTIVE = 'YES' AND DB_PRICE > 10000
```

to retrieve only those records in which the DB_SPORTIVE field is YES and the DB_PRICE field is greater than 10000.

Schematically, the "SELECT" statement built by the Rules Element will look something like this:

```
SELECT field_names FROM table_names WHERE query...
```

where:

- field_names are the fields specified in the "fields and columns" list
- table_names are the names preceding the word WHERE in the Query field
- Query is the string after the word WHERE in the query field.

The query field is also where a SQL join operation is built. A "join" takes the data from two or more tables and unifies them into a single "result" table based on the "WHERE" clause in the SQL statement. The Rules Element sees the result of a join just as it would rows from a single table.

When coding a join in the query field, it's important to remember that the field names are copied "as-is" from the Fields and Properties list into the SQL select statement. In a join, it may not be sufficient to just code a simple field name, since there could be ambiguity in which table fields come from. Consider the following query:

```
CARS, DEALERS WHERE CARS.DB_MODEL = DEALERS.DB_MODEL
```

If DB_MODEL is specified in the fields and properties list, there will be ambiguity since the database manager will not know which table - CARS or DEALERS - to retrieve the field DB_MODEL from. To avoid this problem, DB_MODEL should be specified as CARS.DB_MODEL or DEALERS.DB_MODEL.

Example

The Query - composed of the table names to be accessed and optionally followed by the word "WHERE" and a SQL query clause - is specified in the Query field of the database Retrieve window. The query should NOT be

enclosed in quotes. The following example shows how to retrieve only those records from the CARS table where the `Sportive` field contains YES:

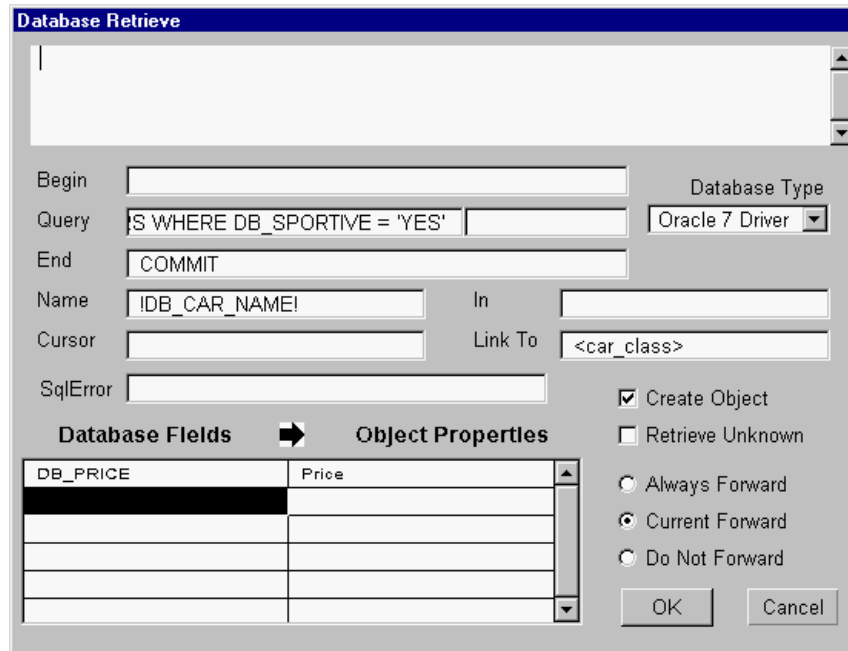


Figure 3-7 Using the Query Field to Retrieve Selected Records

Related Topics

Database Editor Windows
Query Language
Query Arguments
Query

Arguments Overview
Query Example (Sequential Retrieve)
Query Write Operations

Specific database operations and database topics provide more details and examples on how to use the query statement.

Query Field in Write Operations

This section discusses how to build the Query field for write operations. The Query field supplies another level of criteria determining which records will be written by writing to only those records whose fields contain certain values. Two kinds of queries can be used with the Rules Element:

- For relational databases such as Oracle, INGRES, and Sybase, any ANSI-standard SQL query supported by the database may be used. See the appropriate database topic for details.
- For non-relational databases, the Rules Element's own SQL-like query language can be used to filter records. Look up the Query Language topic for more detail.

Query Field

The Query field is used in atomic and grouped write operations. In the case of atomic writes, the query is used to uniquely identify the record(s) to be updated by the write. See the Atomic Write Operations topic for more information on this.

Using queries with grouped write operations is useful when not all the information necessary to identify a record is available in the Rules Element's working memory. Recall that the Name field uses the object name to identify records in the database, but it may be that this is not sufficient to limit the records written to the database.

For example, assume that there are three car objects - `car_1`, `car_2`, and `car_3` - and each object has the properties `Model`, `Model_Date`, `Price`, and `Sportive`. Assume that the `Price` properties have been updated to reflect a sale, but, due to a special promotion, only red cars will be marked down, and therefore only the red car's database records should be updated.

If the car objects had a `Color` property, then an existential pattern matching operation could be used to select only those objects with a `Color` property of `red` to be written. In this example, however, there is no `Color` to do the pattern matching on.

Remember that the Name field constructs a record "key" based on the object name and compares it to selected record fields. There's no way to use the Name field to check for red cars in the database.

However, by including a SQL or SQL-like query in the Query field, you can limit the database records updated to those which have `Red` in the `DB_COLOR` field (assuming, of course, that there is a `DB_COLOR` field in the database), by using a query like this:

```
WHERE DB_COLOR = 'RED'
```

This causes the Rules Element to consider ONLY those records which have a `DB_COLOR` field of `RED`. Note that the conditions specified in the Query are "anded" with any conditions imposed by the Name field. For example, recall that if the Name field is specified as `!DB_CAR_NAME!`, then the following "WHERE" clause would be generated as the object `car_1` was written:

```
WHERE DB_CAR_NAME = 'car_1'
```

Combining this with the query example above, the generated "WHERE" clause would look like this:

```
WHERE DB_CAR_NAME = 'car_1' AND DB_COLOR = 'RED'
```

This has the affect of updating `car_1`'s record ONLY if `car_1` is `red`.

Caution must be exercised when the Create New Record box is checked in the Write window and a query is specified in the Query field. In this case, if no record is found to match the Query and Name criteria, the Rules Element will add a record to the database for the object. However, since, in this example, there's no `Color` property in the cars objects, the `DB_COLOR` field can't be filled in when the record is written. This could generate records whose contents are illogical or invalid.

Specifying Queries for Relational Databases

When writing records to a relational database such as INGRES, Sybase, Oracle, or SQL/DS, the query is handled by the central database manager or server. Therefore, the query can use whatever implementation of the ANSI SQL standard is supported by the particular database being used.

Keep in mind that using specialized features of a given database will mean that the write operation may have to be changed if another database type is used. Generally, if the query uses only those features defined by the ANSI SQL standard, it will be portable across most, if not all, relational database products.

The first thing in the query field must be the table name to be written by the write operation. The name can be in any format legal for the database being accessed.

If no query criteria are to be applied during the write, then nothing except the table name should be specified in the Query field.

The second part of the query field is the "WHERE" clause to be included in the SQL UPDATE statement, and MUST be preceded by the word "WHERE". It is also included "as-is" in the UPDATE statement constructed by the Rules Element.

For example, to write only the records from the CARS table in which the DB_SPORTIVE column contains YES, the query field would contain the following:

```
CARS WHERE DB_SPORTIVE = 'YES'
```

More complex queries can be specified, such as:

```
CARS WHERE DB_SPORTIVE = 'YES' AND DB_PRICE > 10000
```

to write only those records in which the DB_SPORTIVE field is YES and the DB_PRICE field is greater than 10000.

Schematically, the UPDATE statement built by the Rules Element will look something like this:

```
UPDATE table_name WHERE name_column = object_name AND query SET field_name = slot_value, field_name = slot_value, ...
```

where:

- table_name is the names preceding the word WHERE in the Query field
- column_name is one of the column names specified in the Name field between exclamation points (!).
- object_name is the object name (or portion thereof) extracted to be matched against column_name
- query is the string after the word "WHERE" in the query field.
- field_name and slot_value are the "Field and Property" pairs specified in the Write window.

It is NOT possible to use a join operation during a write.

Example

The Query - composed of the table name to be accessed and optionally followed by the word "WHERE" and a SQL query clause - is specified in the

Query field of the database Write window. The query should NOT be enclosed in quotes. The following example shows how to write only those records from the CARS table where the DB_SPORTIVE field contains YES:

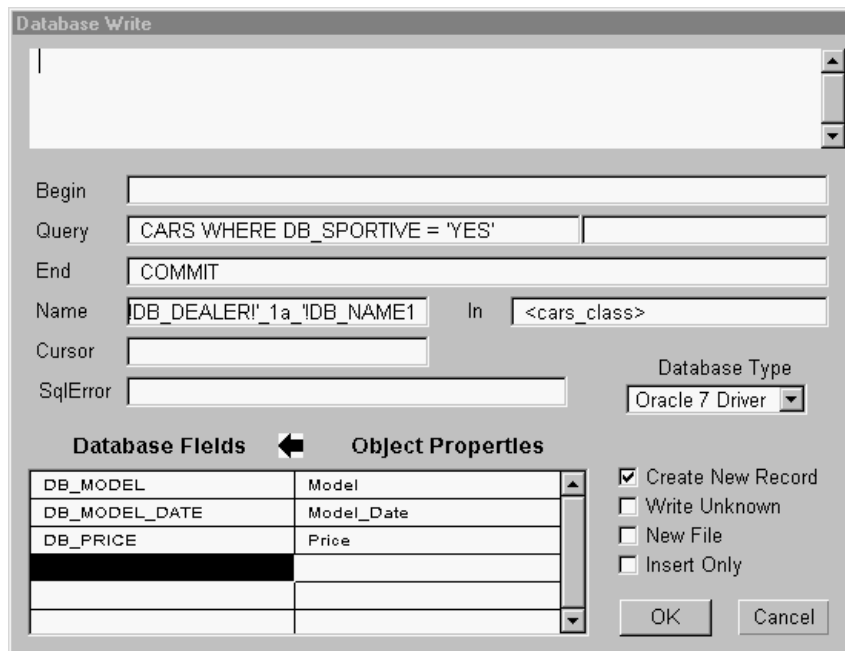


Figure 3-8 Using a Query in a Write Operation

Related Topics

Database Editor Windows Arguments Overview
 Query Retrieve Operations Query Language
 Query Arguments Query

Specific database operations and database topics provide more details and examples on how to use the query statement.

Record Specification for Writes

After the Rules Element selects the slots (object.property combinations) to be written, it writes them out to records (actually, fields within records) in the database. This section discusses how the Rules Element determines which records will receive the data.

Writing by Position

During sequential operations, the Rules Element stores its current position (in the database) in the cursor slot specified in the retrieve or write window. When a sequential write is issued, it writes the record at the position stored in the cursor.

Thus, the logic in the knowledge base determines which records will be written during a sequential write operation. For example, if the knowledge

base issues a sequential write after each read to the database, it will effectively update every record in the database:

- For the first retrieve, the Rules Element will fetch record #1 in the database, and leave the cursor positioned at the beginning of the first record.
- When the sequential write is issued (using the cursor), it will overwrite record #1, and position the cursor at record #2.
- The next retrieve will fetch record #2, and leave the cursor positioned at the beginning of the record.

How to Write by Position

Write by position is supported **ONLY** for sequential write operations. Remember that sequential write is **NOT** supported for most relational databases such as Oracle, Sybase, and INGRES. To specify write by position, you:

- Specify a cursor name in the Cursor field of the Write window.
- Ensure that the cursor value is 0 for the first sequential retrieve or write operation.
- Ensure that the cursor is set to the position where you would like the next record written when the Write is issued.

Specifying a cursor name

You specify the cursor name as a slot name (object.property combination) in the Cursor field of the database write window. This slot must be an "Integer" type.

Ensuring the cursor value is 0 for the first sequential operation

When the Rules Element begins a write operation in which a cursor is specified, it first checks the value of the cursor to determine the type of operation. If the value is 0, it's assumed to be the first sequential read or write; if it's nonzero, it's assumed to hold the position of the next record to be accessed.

It's very important to ensure that the cursor has the appropriate value before the write is issued. Failure to set the cursor properly can result in the Rules Element issuing an atomic write instead of a sequential write, or encountering errors during the write operation.

Ensuring the cursor is set to the record position for subsequent operations

When attempting to add records to a database, or replace existing records, you must ensure that the sequential write is properly coordinated with read operations to ensure that the cursor is set to the proper value. This is done by specifying the same slot name for both the retrieve and write operations.

Writing by Key

During a grouped write, the Rules Element takes the selected objects (actually, object's slots) and writes them to the database in a single operation. To determine which objects will be written to which records, the Rules Element builds a record "key" to identify the record(s) which will receive the object's slots.

The record key is built by taking the object name and comparing it to the appropriate fields in the database records. Records whose field values match the key (or keys) are considered to be a match for the object, and its slots will be written to those records. If no matches are found, a record can optionally be created.

How the object name is compared to the field(s) is very flexible: all of the name can be compared to a single field, part of the name can be compared to a single field, parts of the name can be compared to multiple fields, and so forth.

Simple Keys

As a simple example, assume that there are four objects to be written whose names are HONDA, PINTO, TOYOTA, and BMW. The database records contain a field called DB_MODEL which will be considered the "key" for this write operation. As each object is written, the Rules Element searches the database for a record where the value of the field DB_MODEL matches the object's name. Thus, the HONDA object's slots will be written to the record with the DB_MODEL field of HONDA the CHEVROLET object will be written to the record whose model field contains CHEVROLET, and so forth. Figure 3-9 illustrates this example

Figure 3-9 Using an Explicit Field Name as the Record Key

Complex Keys

As a more complex example, assume the object names are CAR_TOYOTA and CAR_HONDA, but the DB_MODEL fields still contain TOYOTA and HONDA. It's possible to split the object names into two parts: the constant CAR_, and the model name, and have only the model name matched against the DB_MODEL field in the records.

The object name can also be matched across multiple fields. In this case, assume that that object names are composed of the car's model, a constant,

and the car's name: HONDA_is_car_1, TOYOTA_is_car_2, and so forth. The name can be divided into three parts: the model, a constant ("_is_"), and the car name. The model and the name can then be used as "keys", and matched against the DB_MODEL and DB_CAR_NAME fields in the database. Figure 3-10 illustrates this example.

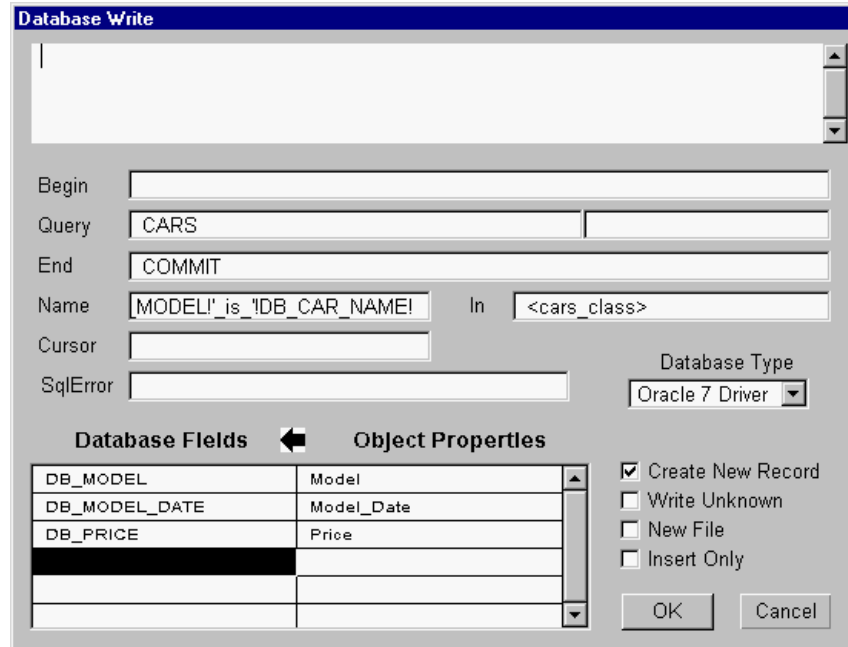


Figure 3-10 Using a Constructed Field Name as the Record Key

Summary

When writing to relational databases such as INGRES, Sybase, Oracle, and Informix, the Rules Element builds a SQL "UPDATE ... WHERE ..." statement to update the proper rows using the "key" values. Using the simplest car inventory example above, SQL statements like the following would be built:

```
UPDATE CARS SET ... WHERE DB_MODEL = 'HONDA'
UPDATE CARS SET ... WHERE DB_MODEL = 'CHEVROLET'
```

For the case where two columns are used as "key" fields:

```
UPDATE CARS SET ... WHERE DB_MODEL = 'HONDA' AND DB_CAR_NAME = 'car_1'
UPDATE CARS SET ... WHERE DB_MODEL = 'CHEVROLET' AND DB_CAR_NAME = 'car_2'
```

Note that any additional WHERE clauses specified in the Query field of the Write window will be appended to these WHERE clauses.

In most cases, there will be a one-to-one correspondence between objects and records in the database. In the case of the cars example, the key would be constructed so that one car object would be written to exactly one database record. If the car database contained only four records - one of each car model - then the simple scheme of mapping the DB_MODEL field directly to the object name would suffice. However, realistically, the DB_MODEL field may not be enough to uniquely identify the records, and a more complex scheme - such as using the model and car name - may be necessary.

It's also possible to have one object written to MANY records. This is done by constructing a key which is not unique to one record. In this case, the object's properties will be written to all records whose field value(s) match the key. For example, in a realistic car inventory, many records would have the same value in the `DB_MODEL` field. If the object names were mapped directly to the `DB_MODEL` field, then each object would be written to multiple records. Thus all of the records for HONDA cars would be updated by the object named HONDA, and so forth. This technique is useful for updating a group of records.

For example, if all the cars of the model TOYOTA were moved to a new location, this type of key could be used to update all the records in a single operation. Obviously, this type of write should specify only the properties and fields which are to be set the same in all records. Writing out properties which are not common to all records - such as `DB_CAR_NAME` - would not be desirable since the `DB_CAR_NAME` in all records would receive the same value!

How to Build Record Keys

When filling in the Write window, you build record keys from the object name by specifying how the Rules Element is to construct the record's name, or key fields from the object name.

You provide the model for constructing the record key(s) in the Name field of the Write window. It is specified as a series of constants (or "roots") and/or field names. The Name field tells the Rules Element how to break up the object name into separate parts to build the record's key, and what fields in the record will be matched against what parts of the key.

The root fields should be enclosed in single quotes, and the field names in exclamation points ("!"). For example, to specify that the entire object name is to be matched against the field name `model` (thus making `model` the key field), you would code the Name as `!model!`.

If the object names were prefixed by the constant `CAR_`, but only the portion of the name following the constant was to be matched against the database field `DB_MODEL`, you would code the Name field as `'CAR_'!DB_MODEL!`.

Multiple fields can be used as record keys: if the object names were composed of the car's model, a constant `_SERIAL_`, and the car's name, then the Name field would be specified as

`!DB_MODEL!'_SERIAL_'!DB_CAR_NAME!`. In this example, the record fields `DB_MODEL` and `DB_CAR_NAME` are the record "keys".

When specifying a Name field which combines constants and/or multiple fields, it is very important that the Name field is unambiguous. For example, a Name field of `!DB_DEALER!'!DB_CAR_NAME!` is ambiguous, since the Rules Element has no way of telling which part of the object name is to go in the `DB_DEALER` field and which is to go into the `DB_CAR_NAME` field.

The Name field must also be accurate: If Name is specified as `'A_CAR_'!DB_MODEL!`, and the object names are all of the form `CAR_model`, then the Rules Element won't be able to match any of the object names against the Name field, and no records will be written. See the Slot Specification for Writes topic for more information on this.

Remember that the Name field is composed of field names, not property names. The field names specified must be present in the records being retrieved, otherwise an error will occur.

Field names which occur in the name field must NOT be repeated in the Fields and Properties list. The field names specified in the Name field are the record's "key", or name, and cannot be changed in the same operation in which they are used to identify the record.

The screenshot shows the 'Database Write' dialog box. The 'Name' field is filled with 'ELI' SERIAL 'IDB_CAR_NAME!' and the 'In' field is filled with '<cars_class>'. The 'Database Type' is set to 'Oracle 7 Driver'. The 'Database Fields' and 'Object Properties' table is visible, with 'DB_MODEL' selected. The 'Create New Record' checkbox is checked.

Database Fields	Object Properties
DB_MODEL	Model
DB_MODEL_DATE	Model_Date
DB_PRICE	Price

Figure 3-11 Filling in the Name Field

Related Topics

Arguments Overview
Name

Create New Records
Slot Specification for Writes

Records Filtering

General

In most transactions, the Retrieve or Write operation does not process all the records stored in the database, but only processes a limited subset. The records are filtered by the transaction. There are two ways by which records can be filtered:

- Records can be filtered by a selection criteria expressed in the Query statement (@QUERY). For example, a query may retrieve only the employee records which have a salary greater than \$4000. This type of filtering is possible only if a query language is available. For relational databases, this query language is typically SQL (or RDO for RDB). For flat database files, you can use the Rules Element Query Language.
- Records can be filtered by the fact that they match a set of existing objects or slots in the working memory of the Rules Element. For

example, a query may retrieve the salary from the employee records for which there is already an employee object (an instance of the employees class) in the Rules Element object base. This type of filtering is controlled by the In List (@ATOMS) and the slots/properties lists (@SLOTS/@PROPS). This type of filtering can be performed only if the Create Object setting (@FILL=NEW;) is disabled.

Related Topics

Query	In List
Create Objects	Arguments Overview
Query Language	

Retrieve Operator

The `Retrieve` operator is used in rules and methods to read information from a database or spreadsheet.

Operands

The `Retrieve` operator takes two operands:

- The first operand is either a string constant or an interpretation to a string constant specifying the name of the file containing the database to be queried or the login name/access string for a DBMS.
- The second operand consists of a series of arguments defining the specific retrieval operation to be performed.

Arguments

The second operand may include the following arguments:

@TYPE	Type of database (creator software and file format)
@BEGIN	Command string for opening transaction
@END	Command string for closing transaction
@QUERY	Command string for querying database
@ERROR	Slot name to trap database error message
@ARGS	Argument list for query command
@ATOMS	List of objects or properties affected
@NAME	Correspondence between records and objects
@FIELDS	List of field names to retrieve from
@PROPS	List of properties to retrieve to
@SLOTS	List of slots to retrieve to
@FILL	Create new objects
@CREATE	Classes or parents to link new objects to
@UNKNOWN	Retrieve UNKNOWN values
@FWRD	Forward retrieved values
@CURSOR	Current position for sequential retrieval

When entering a `Retrieve` action in the Rule Editor or Method Editor, clicking in the space for the second operand displays the Database Editor window for specifying the retrieval arguments interactively, rather than by explicitly typing them in as listed above.

Note: It is valid to have an empty second operand. When this occurs, the Rules Element will determine the type of database from the filename extension specified in the first argument, and will default to the SYLK type if no extension is specified. Only simple spreadsheet files can be accessed in this case. This operating mode has been maintained to ensure compatibility with earlier versions of the Rules Element.

Effect

The requested information is retrieved from the specified database to the Rules Element the Rules Element knowledge base for further processing.

Result

When used in a condition on the left-hand side of a rule, the `Retrieve` operator always produces a `TRUE` result, even if no records are retrieved satisfying the given query. The only exception is if an error occurs while attempting to open the database or transmit the query, in which case the result is `FALSE`.

Related Topics

Access String	Left-Hand Side Retrieves
Access String Specification	Right-Hand Side Retrieves
Arguments Overview	Order of Sources Retrieves
Database Editor Window	If Change Retrieves
Interpretations @V(...)	

Look up the following topics in Chapter One, “Application Development Features” for information related to the `Retrieve` operator.

Rules	Classes
Methods	Properties
Actions	String Constants
Objects	

Retrieve Unknown - (@UNKNOWN)

Usage

The `Retrieve Unknown` setting is meaningful in all types of transactions. It controls whether or not `UNKNOWN` values should be retrieved by the transaction.

This setting is specified with the Retrieve Unknown check button in the Database Editor windows. In the text form of the knowledge base, it is saved as:

```
@UNKNOWN=TRUE ;
```

or

```
@UNKNOWN=FALSE ;
```

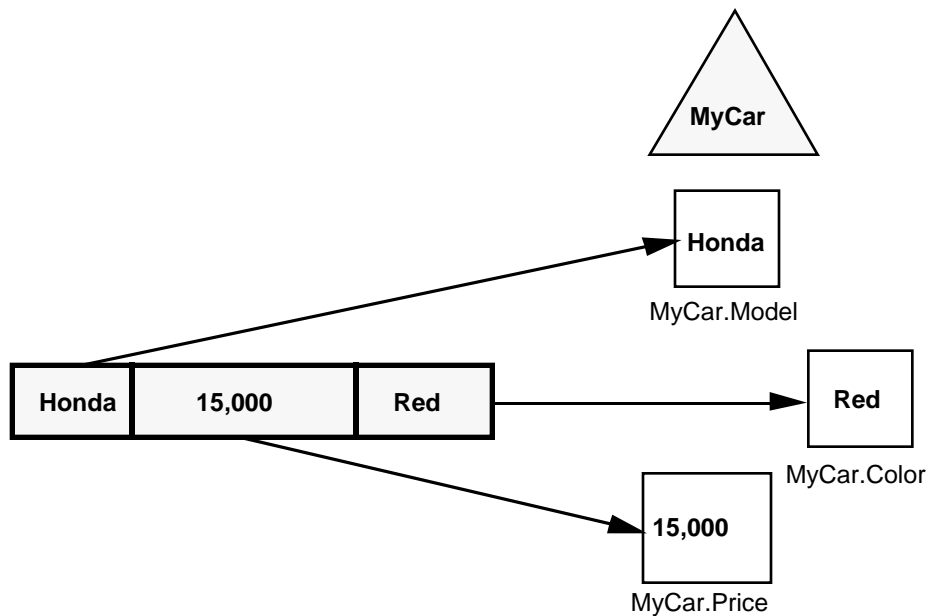
Related Topics

Database Editor Windows
Retrieving from Databases
Arguments Overview

Retrieving from Databases

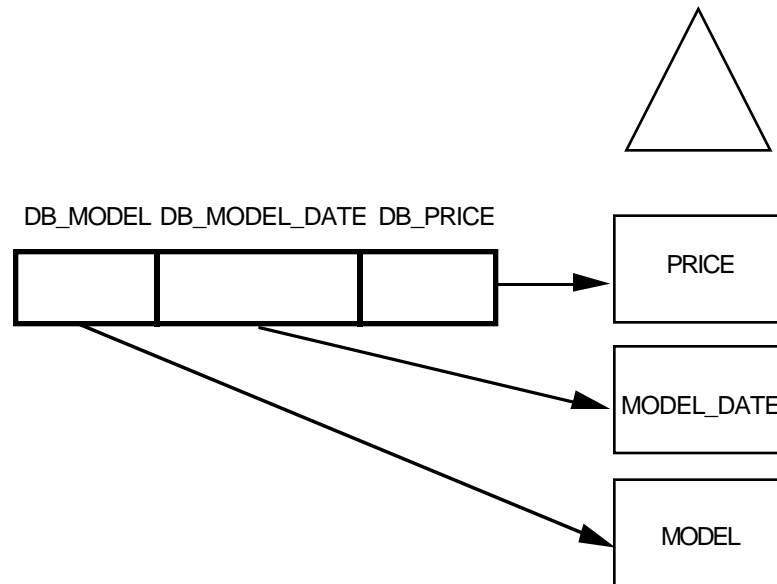
General

During most retrieve operations, the Rules Element selects a single object to receive each record's fields, and the fields are read into the object's slots. Thus, the contents of a record are represented by an object, and the fields in the record are represented by the object's property slots. This has the affect of transforming the record-field relationship into an object-property relationship in the Rules Element's working memory.



For example, take the case of a car inventory file. Each car is represented by a record with the fields `DB_MODEL`, `DB_MODEL_DATE`, and `DB_PRICE`. In the knowledge base, a car is represented by an object with the properties `Model`, `Model_date`, and `Price`. The Retrieve operation in the

knowledge base specifies the mapping between the record's fields and Rules Element properties



Database Fields	Object Properties
DB_MODEL	Model
DB_MODEL_DATE	Model_Date
DB_PRICE	Price

In this example, all of the fields from a car's record are mapped into one object's slots, and thus a car's record is "transformed" into a car object.

Depending on the type of retrieval, records can be retrieved one by one and mapped into the same object, or many records retrieved and mapped into many different objects. In either case, as the records are retrieved, the Rules Element is capable of either updating existing objects, or creating new objects to hold the records.

For example, the car records could be retrieved one by one into the same car object, or many cars records could be retrieved at once into many different car objects.

With sequential and atomic retrieval, it's also possible to retrieve a record's fields into slots belonging to two or more objects, in effect "scattering" a record's contents across several different objects.

Of course, it's not always necessary to retrieve all the records in the external file or database. The Rules Element therefore provides several ways of filtering the records which are actually read into its working memory.

This filtering occurs in three stages:

- A SQL or SQL-like query can be used to select a subset of the records from the database based on the data in the record fields themselves.

- An object or object's slots (object.property combinations) are selected to hold the record's fields.
- Existence filtering determines if the selected object exists, and if it does, checks to see if it exists in a specified list of objects or classes. If it doesn't exist, or doesn't exist in the list, the record can either be bypassed or a new object created to hold it.

Related Topics

Databases	Spreadsheets
Grouped Retrieve	Sequential Retrieve
Atomic Retrieve	Retrieve Operator
Query Retrieve Operations	Existence Filtering Operations
Object Names In Retrieve Operations	Slot Specification for Retrieves
String to Numeric Conversion	Retrieve Unknown
Create Object	Debugging Operations
Forwarding Strategy	Formats

Return Errors

Like all Rules Element operations, retrieve and write return a "true" or "false" value depending on the results of the operation.

Flat-Files

Retrieve and Write operations always return "true" unless an error occurs. For flat-file type databases such as spreadsheets, NXPDB, NXP, and DBASE3 files, some of these errors include:

- The file could not be found
- An operating system error occurred while opening the file
- You don't have the authority to access the file
- The file's format was invalid for the database type
- Syntax error in the Query field

Relational Databases

For relational databases such as Oracle, Sybase, INGRES, Oracle, Informix, possible errors include:

- The account specified for the access was rejected by the database
- The table name(s) specified in the Query argument was invalid
- The syntax of the query was invalid for the database
- A column name specified in the query did not exist
- An operating system or database error occurred

It is especially important to note that for all database types, a "record not found" condition is NOT considered an error, and therefore will not invalidate the condition on the LHS of a rule. Thus, a retrieve or write can

return "True" but NO records will have been read or written. Examples of when this can occur include:

- No records met the criteria of the Query argument
- During a retrieve, no records could be mapped to existing object names and "fill" was specified as "no", therefore no new objects could be created and no rows were retrieved.
- During a write, no objects could be mapped to existing records and "fill" was specified as "no", therefore no new records could be created and no records were written.

When designing your knowledge base, you should ensure that it can handle a condition where no records are accessed, yet a "True" condition is returned by the Retrieve or Write operation.

Related Topics

Databases	Spreadsheets
Retrieve Operator	Write Operator
Left-Hand Side Retrieves	Query Argument
Left-Hand Side Writes	Access String
Debugging Operations	

Right-Hand Side Retrieves

Usage

A retrieve statement can also be used in the RHS of a rule or method, but here it's not as useful because it's impossible to test if the Retrieve operation failed, and therefore if there is any valid data to process.

When the Rules Element begins a retrieve operation, it gets the database access string from the first argument of the retrieve statement.

Related Topics

Arguments Overview	Retrieve Operator
Access String	Atomic Retrieve
Sequential Retrieves	Group Retrieve
Left-Hand Side Retrieves	Atomic Retrieve Example
Retrieving from Databases	

Right-Hand Side Writes

Usage

In the right hand side of a rule or method, a write statement is usually used to reflect the consequence of a hypothesis being found "true" in a database. In the case of the car inventory example, if the LHS of a rule determines that a car was sold, then the RHS of the rule could contain a write statement to update the inventory.

Remember that even if the write fails due to an error and returns "False", the RHS will continue execution until all RHS statements have been executed.

When the Rules Element begins a write operation, it gets the database access string from the first argument of the write statement.

Related Topics

Arguments Overview	Write Operator
Access String	Atomic Write
Sequential Write	Group Write
Left-Hand Side Writes	If Change Writes
Order of Sources Writes	Writing to Databases

Sequential Retrieve

General

Sequential retrieval can be used with both flat-file databases and relational databases such as INGRES, Sybase, and Oracle.

The sequential retrieve operation reads the fields from multiple records, one record at a time, into slots in the Rules Element's working memory. The slots (object.property combinations) usually all belong to the same object, but it's also possible to read the fields into slots belonging to two or more objects.

Typically, a knowledge base will use a sequential retrieval to read a record's fields, do some reasoning over the record, "loop back" to retrieve another record, reason over it, and so on. It's also possible to include a sequential write in this loop (for some database types) to write out an updated copy of the record after each reasoning step.

For example, a sequential retrieval could be used to read each record from a "CARS" database into an object's properties, compute a discounted price for the car, and write out an updated record to the database. In this example, each record is processed independently of the next one.

Sequential retrieves require that you provide the logic in your knowledge base to "loop" thru the retrieve until all the records have been retrieved. One approach is to create rules like the following:

- Rule #1 tests the value of the cursor in the LHS to ensure that it's not negative.
- If the cursor isn't negative, Rule #1 issues a Retrieve (in the LHS or RHS) to retrieve the next record's fields into a fixed set of slots.
- Subsequent rules process the slots.
- When the record has been completely processed, the hypothesis of Rule #1 is reset, forcing the next record to be retrieved.

The processing associated with the record can also include a sequential write (using the same cursor slot), which will update the record just retrieved. Remember however, that sequential writes are NOT supported for most database types.

Specification

Sequential retrieves are recognized by the fact that a Cursor slot is provided in the database retrieve window, and it has a positive (0 is defined as a positive number) value when the Retrieve is issued.

For relational databases, the cursor must be set to 0 for the first retrieve, and the Rules Element set to an arbitrary positive number for subsequent retrieves. When all the records have been retrieved, the cursor will be set to -1. The cursor's value must NOT be changed by the knowledge base once the retrieve begins--doing so will cause errors and/or unpredictable results.

For flat-file databases, the Rules Element will read the "Cursor+1"-th record in the database. For example, if the cursor slot has a value of 23 when the retrieve is executed, then the 24th will be retrieved.

A sequential retrieve does not necessarily have to retrieve all the records from the database. It is possible to limit which records are retrieved by supplying a query with the retrieve. For relational databases, you can use any query accepted by the database manager (usually an ANSI SQL statement), for flat-file databases, you can use the Rules Element Query Language to filter the records.

If no records meet the query criteria, then the cursor will be set to -1 on the first retrieve.

A sequential retrieve reads the record fields into specific slots which already exist when the retrieve is issued.

Fields

To build a sequential retrieve, complete the Retrieve screen in the Database Editor window as follows.

- If the Retrieve is to a relational database such as Oracle, Sybase, or INGRES, ensure that the Cursor slot specified in the Retrieve window is 0 before the first retrieve is executed.
- Usually, in the LHS of the rule issuing the Retrieve, a test is specified to ensure that the Cursor slot has not gone negative, which indicates that the last record has been retrieved.
- Specify Retrieve as the operator in the LHS or RHS of the rule.
- As the first operand of the Retrieve, specify the database access string if a relational database is being accessed. If a flat file database such as NXPDB or DBASE III is being accessed, specify the file name. See the Access String Specification Topic for more information.
- In the database Retrieve window, click on the appropriate selection in the Database Type field for the database being retrieved from.
- The Begin field should contain whatever is appropriate for your database. See the Beginning Database Operations topic for more information. Flat-file databases use this field to specify a range name, see the Begin topic for details.
- For a relational database, specify the table name to be accessed in the Query field. If you want to limit the records retrieved by the retrieve, you can also include a SQL query (for relational databases) or a Rules Element SQL-like query (for flat file databases) in this field. See the Query Retrieve Operations topic for more information on filling in the Query field.

- The End field should contain whatever is appropriate for your database to end a transaction.
- The slot names (object.property combinations) to receive each record's fields are specified explicitly. See the Slot Specification for Retrieves topic for more information.
- The Cursor field should contain the name of the slot to be used as the cursor for this retrieve operation. This slot must be of the integer type, and MUST have a value of 0 when the retrieve is issued from a relational database. The slot name may be specified as "object.property" or just "object", which is shorthand for "object.Value".
- In the Database Fields column, specify the names of the database fields to be retrieved. In the corresponding Object Properties column entries, specify the property slots into which the fields should be retrieved. See the Slot Specification for Retrieves topic for more information.
- The Create Object option must be left unselected. Only grouped retrieves can be used to create objects.

Related Topics

Cursor Slot Specification	Query Retrieve Operations
Database Editor Windows	Slot Specification for Retrieves
Object Names In Retrieve Operations	Query Example
Sequential Retrieve Example	Query Language

Also, look up individual arguments and your database type for more detailed information.

Sequential Write

General

Sequential Write operations can be used **ONLY** with RDB RDO or with flat-file databases such as NXPDB and DBASE III. It can **NOT** be used with relational databases (other than RDB RDO).

The Sequential write operation writes a set of slots into database fields one record at a time. The slots (object.property combinations) usually all belong to the same object, but it's also possible to write slots belonging to two or more objects to each record. Each record is written from the same set of slots which are presumably updated in the logic between the executions of the Write statement.

Typically, a knowledge base will use a sequential write to rewrite updated records during a sequential read operation. For example, a knowledge base would use a sequential read to reach a record, rules would reason over its contents, possibly change some slot values, and a sequential write would replace the record in the database.

Sequential writes can also be used in a standalone fashion (not in conjunction with a sequential retrieve), in which case the Cursor field is used to position the database to the correct record before each write operation.

A sequential write requires that some logic be built around the write operations to support them. The amount of logic required depends on whether the write is used in conjunction with a sequential read.

If the sequential write is NOT used in conjunction with a sequential read, then the logic in the knowledge base must set and maintain the cursor's value to correspond to the record number to be written.

If the write is associated with a sequential read, then the read operations will take care of setting and maintaining the cursor value once the retrieve begins. See the Sequential Retrieve operations topic for more information.

Specification

Sequential writes are recognized by the fact that a Cursor slot is provided in the database retrieve window, and it has a positive (0 is defined as a positive number) value when the Write is issued.

The Rules Element will write the Cursor-th record in the database. For example, if the cursor slot has a value of 23 when the write is executed, then record 23 will be written.

A sequential write cannot add records to a database, it can only update existing records.

Fields

To build a sequential write, complete the Write screen in the Database Editor window as follows.

- Ensure that the cursor slot's value is a positive value (0 is considered positive) before the write is issued.
- Specify Write as the operator.
- As the first operand of the Write specify the file name to be accessed. See the Access String Specification topic for more information.
- In the database Write window, click on the appropriate selection in the Database Type field for the database being written. Remember that sequential writes can NOT be used with most relational databases.
- The Begin and Query fields should be left blank.
- The End field should contain whatever is appropriate for your database to end a transaction. For almost all relational databases, either "COMMIT" or "COMMIT RELEASE" should be specified. See the Ending Database Operations topic for more information.
- The Name field may be left blank or may contain an explicit object name whose property slots will be written to each record's fields. See the Slot Specification for Writes topic for more information.
- The Cursor field should contain the name of the slot to be used as the cursor for this write operation. This slot must be of the integer type, and MUST have a positive value when the retrieve is issued. The slot name may be specified as "object.property" or just "object", which is shorthand for "object.Value".
- The In and Link to fields should be left empty.

- In the Rules Element Properties column, specify the property slots which are to be written to the fields in the database. In the database fields column, specify the corresponding field which is to receive each property slot. See the Slot Specification for Writes topic for more information.
- The Create New Record option must be left unselected. Only grouped writes can be used to create records.

Related Topics

Cursor Slot Specification

Sequential Retrieve

Also, look up individual arguments and your database type for more detailed information.

Slot Specification for Retrieves

As the Rules Element retrieves a record or records, it takes the data from the fields and places it in the property slots of one or more objects. Usually, a given record's fields are almost always read into a single object's slots - thus preserving the record and field relationship as objects and properties.

Remember that property slots are identified as "object.property", where "object" is the object name, and "property" is the property name. The property names are always specified explicitly in the database Retrieve window (in the right hand side of the Fields and Properties list). The object names can be determined in a number of ways, including from the data in the records themselves. This section describes how object names are built during Retrieve operations.

Using Explicit Object Names

There are two ways to specify the slots which will receive the fields from the records:

- You explicitly state each "object.property" name in the right hand side of the fields and properties list, opposite the corresponding field specification. The Name field is left empty.
- You list only the property name(s) in the right hand side of the double column list, and specify the object name in the Name field. As the records are retrieved, the Rules Element uses these fields together to form the slot names.

Both techniques are equally valid, and in almost all circumstances, there's no advantage to using one technique over another. One exception is that listing the "object.property" combinations explicitly allows you to split a record's fields among two or more objects.

The following illustrations show how to use each of these techniques to retrieve the fields DB_MODEL, DB_CAR_NAME, and DB_PRICE in object MyCar's Model, Name, and Price properties.

The screenshot shows the 'Database Retrieve' dialog box. The 'Query' field contains 'CARS' and the 'Database Type' is 'Oracle 7 Driver'. The 'Cursor' is 'Cursor_Object.Cursor_Slot' and 'Link To' is 'CARS_CLASS'. The 'Object Properties' list is populated with the following data:

Database Fields	Object Properties
DB_MODEL	MyCar.Model
DB_MODEL_NAME	MyCar.Model_Name
DB_PRICE	MyCar.Price

Other options include 'Create Object', 'Retrieve Unknown', and radio buttons for 'Always Forward', 'Current Forward', and 'Do Not Forward'. 'OK' and 'Cancel' buttons are at the bottom right.

Figure 3-12 Using Slot Names in Properties List

The screenshot shows the 'Database Retrieve' dialog box. The 'Query' field contains 'CARS' and the 'Database Type' is 'Oracle 7 Driver'. The 'Name' field contains 'MyCar'. The 'Cursor' is 'Cursor_Object.Cursor_Slot' and 'Link To' is 'CARS_CLASS'. The 'Object Properties' list is populated with the following data:

Database Fields	Object Properties
DB_MODEL	Model
DB_MODEL_NAME	Model_Name
DB_PRICE	Price

Other options include 'Create Object', 'Retrieve Unknown', and radio buttons for 'Always Forward', 'Current Forward', and 'Do Not Forward'. 'OK' and 'Cancel' buttons are at the bottom right.

Figure 3-13 Using Property Names Only in the Properties List

In the first example, we have listed the "target" slots MyCar.Model, MyCar.name, and MyCar.Price explicitly in the right hand side of the fields and properties list, across from their corresponding fields DB_CAR_NAME, DB_MODEL, and DB_PRICE.

The second example shown accomplishes the same thing, except that only the properties are listed in the fields and properties list, and the object name - MyCar - is listed explicitly in the Name field.

Using Constructed Object Names

Constructed object names are used only with grouped retrieve operations. To specify the slots to receive the fields, you list only the property name(s) in the right hand side of the double column list. As the records are retrieved, the Rules Element combines the generated name for the object with these property names to form the actual slot names to receive the records' data.

The following illustration shows how to build object names from record data. The object names are formed using the DB_MODEL and DB_CAR_NAME fields. The fields DB_MODEL, DB_MODEL_DATE, and DB_PRICE are retrieved into the property slots Model, Model_Date, and Price.

The screenshot shows the 'Database Retrieve' dialog box. It has several input fields: 'Begin', 'Query' (containing 'CARS'), 'End', 'Name' (containing 'MyCar'), 'Cursor', 'Link To' (containing 'CARS_CLASS'), and 'SqlError'. A 'Database Type' dropdown is set to 'Oracle 7 Driver'. Below these fields is a table with two columns: 'Database Fields' and 'Object Properties'. The table contains the following entries:

Database Fields	Object Properties
DB_MODEL	Model
DB_MODEL_DATE	Model_Date
DB_PRICE	Price

Below the table are several options: 'Create Object' (checked), 'Retrieve Unknown' (unchecked), and three radio buttons: 'Always Forward', 'Current Forward' (selected), and 'Do Not Forward'. 'OK' and 'Cancel' buttons are at the bottom right.

Figure 3-14 Building Slot Names from Record Data

Related Topics

Retrieving Databases

Name

Object Names In Retrieve Operations

Also see the Grouped Retrieve example in Appendix A, "Database Integration Examples" for further illustrations of the Name field.

Slot Specification for Writes

When a write operation is requested, the Rules Element first selects the slots (object.property combinations) which are to be the source of the write operation. There are two ways to specify the slots to be written:

- As an explicit list of "object.property" combinations
- As a list of object names or classes along with a list of properties to be written from them.

This section describes these techniques in detail.

Using Explicit "Obj.Prop" Combinations

For sequential and atomic write operations, you specify a list of "object.property" combinations to be written to each record. In this case, the fields are always written from the same slots. Usually, all of the slots are from the same object, but it's also possible to specify slots from two or more different objects.

With atomic write operations, the use of this technique is quite simple: the logic in the knowledge base fills in the slots, and the slots are written to the fields in the database. A slot name can be specified more than once in the list.

The slots to be written can be specified by listing them as explicit "object.property" combinations, or by specifying the object name and listing the properties in the Fields and Properties list. Both techniques are equally valid. If slots from two or more different objects are to be written, the first technique must be used.

Specifying explicit object.property combinations

To specify explicit slot names, list them in the Properties column of the Fields and Properties list, opposite the fields which the slots will be written to. The following example shows how the slots in object `MyCar` could be written to the database:

The screenshot shows the "Database Write" dialog box. It contains several input fields: "Begin", "Query" (with "CARS" entered), "End", "Name", "In", "Cursor" (with "Cursor_Object.Cursor_Slot" entered), and "SqlError". The "Database Type" is set to "Oracle 7 Driver". Below these fields is a table with two columns: "Database Fields" and "Object Properties". The table has three rows: "DB_MODEL" (MyCar.Model), "DB_MODEL_DATE" (MyCar.Model_Date), and "DB_PRICE" (MyCar.Price). To the right of the table are four checkboxes: "Create New Record" (checked), "Write Unknown", "New File", and "Insert Only". "OK" and "Cancel" buttons are at the bottom right.

Database Fields	Object Properties
DB_MODEL	MyCar.Model
DB_MODEL_DATE	MyCar.Model_Date
DB_PRICE	MyCar.Price

Figure 3-15 Using Slot Names in the Properties List

Using "Obj.Prop" Combinations

There are two ways to specify the slot names:

- As explicit "object.property" combinations.
- As an object name and a list of properties.

Specifying an object name and list of properties

To use this technique, specify the object name (enclosed in single quotes) in the Name field of the Write window, and the properties in the Properties column of the Fields and Properties list opposite the corresponding database fields. The following examples show how to write MyCars's slots to the database.

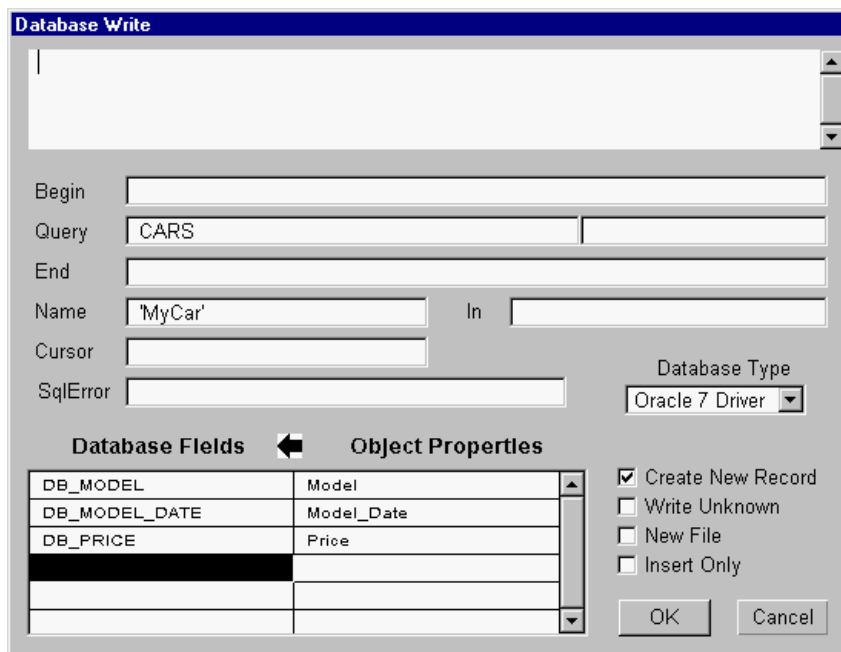


Figure 3-16 Using Property Names Only in Properties List

Related Topics

Name Writing to Databases
Record Specification for Writes Arguments Overview

Also see the Grouped Write example in Appendix A, "Database Integration Examples" for further illustrations of the Name field.

Slots List - (@SLOTS)

General

The slots list can be specified in all types of transactions except operations on spreadsheet files. This list is edited in the right part of the double list box at the bottom of the Database Editor windows.

In the case of sequential or atomic transactions, the list is a list of slots (separated by commas), and it is prefixed by the @SLOTS keyword.

```
@SLOTS=list of slots;
```

This is very similar to the Properties List (@PROPS) described for that topic.

Related Topics

Database Editor Windows
Arguments Overview

Slot Specification for Writes
Slot Specification for Retrieves

Spreadsheets**General**

The spreadsheet files have formats associated with spreadsheet programs such as Lotus 1-2-3, EXCEL, and the Rules Element's own spreadsheet format (also referred to as "NXP"). In these formats, each spreadsheet cell is treated as a unique data item, completely unrelated to other cells in the file.

In a spreadsheet, individual cells are usually addressed by row and column like "A1", "C23", "KK16". This works well in the context of a user interface, but it is not very convenient when it comes to identifying data items in a file. Not only is there no "dictionary" of which cells represent which data items, there is the problem that insertion of a row or column shifts the row-column coordinates of many other cells, and invalidates any references to their old positions (inserting a column between column "A" and "B" means that what was in position "B1" is now in "C1", etc);

Therefore, to use a spreadsheet file with the Rules Element database interface, the cells which will be accessed from the Rules Element must have a Name or "definition" attached to them. This Name is stored by the spreadsheet software with the spreadsheet, and provides a consistent reference for a particular datum no matter how its position changes.

Although the ability to read and write spreadsheet format files is useful for accessing existing information from LOTUS 1-2-3 or EXCEL files, it is not so useful as an application "database". The spreadsheet's simple nature makes it difficult to group data together into logical entities. For example, there is no built in way to state that "cells A1, A2, A3, and A4 represent CAR_1's price, model, model date, and sportiness", and "cells B1, B2, B3, and B4 are CAR_2's price, model, model date... ", and so forth.

Related Topics

Retrieving from Databases
Query Language
SYLK
Rules Element Flat-File Formats

Writing to Databases
WKS
Arguments Overview

SqlError - (@ERROR)**Usage**

The database server that you initiate transactions with may generate error messages or error numbers that you can trap at runtime. The SqlError field of the Database Editor window lets you specify a slot that you create for this purpose. If an error occurs, the message generated is stored as the value of the slot and the transaction is immediately halted.

Your knowledge base might use an if change method to test the value of the error slot each time its value changes. At runtime, if the database returns either an error number (the slot should be of type Integer) or an error message (the slot should be of type String), the transaction is immediately halted, and the inference engine automatically sets a left-hand side Retrieve or Write condition to FALSE. If no error slot is specified, error messages that are generated at runtime can be viewed in the Transcript window that you enable.

In text knowledge bases, the field list is saved as a list of quoted strings. The formal syntax is:

```
@ERROR=slot name
```

Note: If the slot name is specified in the Database Editor window, the Rules Element automatically creates the slot for the knowledge base.

Related Topics

- Debugging Operations
- Database Editor Windows
- Arguments Overview
- Retrieving from Databases
- Writing to Databases

For precise information on what is allowed for a given database type, look up your database type.

String to Numeric Conversion {x}

General

The "{x}" syntax is used with relational database queries to provide a "hint" to the Rules Element as to the datatype of the corresponding database field.

Depending on the particular database interface being used and the current availability of the database server and table(s) being accessed, the Rules Element has some, little, or no knowledge of the datatypes of the fields being referenced (retrieved or written) in the database table. In particular, the problem being addressed with this syntax is the case where numeric field values are not being provided without the quotes typically associated with strings. Some databases (e.g. ORACLE) will automatically do most string to numeric conversion. Some of the Rules Element database interfaces (e.g. SYBASE) have some understanding of the Rules Element property type and will generally do the right thing with fields (quoting as appropriate). Other databases need some help, though.

This syntax is used immediately before the database field name in the Field list or in the Name field, with "x" set to be "S" for string, "F" for float, or "I" for integer (i.e. numeric). Only a single character is permitted, and it must be exactly as specified (it is case-sensitive). This syntax should only be used when, for example, use of the Transcript indicates an inappropriate use of quotes by the Rules Element.

Example

As an example, suppose we are using the SYBASE database interface and have two Rules Element properties of type string (`str_money` and `str_int`) that we wish to write into two SYBASE fields of type money and int with field names of `db_money` and `db_int`, respectively. For a normal transaction involving integers or strings, the SYBASE database interface would not need the "{I}" syntax, but in this case we are dealing with money (a type unknown to the Rules Element) and a string we are forcing into an integer field. Using this syntax, the properties and fields lists would look like:

```
@PROPS= str_money,str_int;
@FIELDS= "{I}db_money","{I}db_int";
```

In a similar manner, staying with the SYBASE example, the `db_int` field may actually be part of the object name derived from the Name field as in:

```
@NAME= "'root_!{I}db_int!";
```

In this case, the Rules Element is obtaining the value of `db_int` from a string (part of the object name) and would normally provide the value inside quotes, which SYBASE would not accept. By using the "{I}" syntax again, we have forced a numeric handling.

In many cases, this additional syntax is not required, and it should only be used where the Rules Element is obviously providing a form that the database server will not accept. The database topics contain additional details for the various Rules Element database interfaces on when and where this syntax is required.

Related Topics

- Database Editor Windows
- Formats
- Arguments Overview

SYBASE

SYBASE is the relational database product of SYBASE, Inc. The query language of SYBASE is the standard SQL (Structured Query Language) language. This section assumes familiarity with the SQL language and the SYBASE product.

The Rules Element SYBASE database interface is available as a separate package. An installation guide is provided with the software. It contains all the information required to configure the system and install the database interface.

The basic logic controlling the transactions has been described under the Retrieve and Write topics in this chapter. This part will explain how the SQL queries are constructed.

Database Access String

As explained in the Access String topic in this chapter, the first argument of the Retrieve or Write operators contains the information required to establish the connection with the database. In order to connect with the

SYBASE database server, you must specify the user name and password with which to connect. You may additionally be required to specify a server name and database. You may optionally specify a host name and application name. The correct order for specifying these connection parameters is as follows:

```
"username password hostname severname applicationname
database"
```

You must not skip parameters within the connection string. If you need to, use a dummy name to supply a connection parameter that is not used, but do not skip a parameter or replace one by blanks. For example, the above connection parameters might take the following connection string:

```
"scott tiger hyperion SYBASE_HYPERION MyApp customerdb"
```

In this example, the application name `MyApp` was supplied as a dummy placeholder. Each parameter must be delimited by a blank space.

You cannot be connected to several accounts simultaneously. You can, however, close a connection by issuing a `RELEASE` statement (see End string description below) and open a connection to another account afterwards.

Query Syntax

Begin and End strings

In these strings, you can specify any valid SQL statement which will be sent to the DBMS server. If you want to send several SQL statements, you must separate them by a semi-colon character (;).

The Rules Element recognizes the special word `RELEASE` in the End statement because it needs to be processed differently by the SYBASE connection module. If `RELEASE` is found, the Rules Element closes the connection with the database.

Usually, the Begin statement is left empty for Retrieves. In the case of a Write, however, the Begin statement must be of the form:

```
@BEGIN= "begin transaction transaction_name";
```

where `transaction_name` is a name of the user's choosing. Also, for a Write operation, the End statement will typically be one of the following:

```
@END= "commit transaction";
@END= "rollback transaction";
```

depending on whether the actions performed during the transaction are to be kept or discarded, respectively. By default, the Rules Element will do a rollback when a Restart Session is done.

Another frequently used Begin statement is

```
@BEGIN= "use database_name";
```

to select a database other than from the default database area.

Query string

The query string contains one or several table names followed by an optional where clause.

Let us take an example. Our database contains two tables:

- `employees` with the fields `emp_id`, `name`, `dept_id`, `salary` and `bonus`.
- `departments` with the fields `dept_id`, `name`, `budget`.

You can retrieve all the employee records with the following query:

```
@QUERY= "employees";
```

Note: In the Database Editor, you should not enclose your string in double quotes. You should type only the word `employees`.

You can express complex queries such as:

- (a) @QUERY= "employees where salary > 3000";
 (b) @QUERY= "employees, departments where salary > 3000 and employee.dept_id = department.dept_id";

In the second case (b), the query will join the two tables `employees` and `departments`.

The query string is not sent as is to the DBMS server (it is not a valid SQL statement). The actual SQL query is built in the following way:

- If a Name is specified (grouped queries), the Rules Element extracts the `field1` and the optional `field2...field5` information from the Name (see Name topic for details).
- Then the Rules Element builds the SELECT statement:

```
SELECT field1, field2,...,field5, list_of_fields FROM query_string
```

where `list_of_fields` is the list of fields specified in the left part of the double list box of the Database Editor (@FIELDS).

The resulting string would be the string used with the "isql" program. SQL displays the results of the query on the terminal but the Rules Element needs to assign the retrieved values to some internal variables. Let us consider our example query string (b). If the name slot of our Database Editor contains 'emp_'!emp_id!, and the fields list contains the three properties `name`, `employees.dept_id` and `salary`, then the following string will be sent to the SYBASE server:

```
SELECT emp_id, name, employees.dept_id, salary FROM employees, departments
WHERE salary > 3000 and employee.dept_id = department.dept_id
```

You must fully specify field names which are present in more than one relation. In our example, `dept_id` must be prefixed by a table name (even if the two tables contain the same value for this field as a result of our join operation).

You can use the full power of the SQL language and specify expressions instead of field names (i.e. write `salary + bonus` instead of `salary`) as long as the SQL string which will be generated is a valid SELECT statement. The Transact-SQL User's Guide and the Transact-SQL Commands Reference manual provide detailed information on SQL.

Writing parameterized queries

You can use either the @V(obj.prop) special syntax or the query argument box to parameterize your queries. Our previous example can be transformed as follows:

```
@QUERY= "employees, departments where salary > @V(@SELF.amount) and
employee.dept_id = department.dept_id";
```

or

```
@QUERY= "employees, departments where salary > :v1 and employee.dept_id =
        department.dept_id";
@ARGS= SELF.amount;
```

Note: SELF and interpretations are allowed in the right part of the fields/properties list box (@SLOTS) in the case of sequential or atomic queries (grouped queries use a list of properties, not slots). SELF is allowed only if the query is placed in methods.

Update and Insert statements

UPDATE and INSERT statements are constructed in a similar way. INSERT statements are generated only if the Create New Record option is selected and will concern only the objects specified in the In list which do not already have a matching record in the database.

The UPDATE statement is generated as follows:

```
UPDATE tables_from_query_string SET list_of_fields/values WHERE
[field1 = value_of_field1 [AND field2 = value_of_field2]...] [AND]
[where_clause_from_query_string]
```

The square brackets indicate optional strings. Let us take our example (a) and suppose that the salary field needs to be updated and that the Name cell contains 'emp!emp_id!. The resulting SQL statement will be:

```
UPDATE employees SET salary = 5000 WHERE emp_id = '104' and salary > 3000
```

Note: In this example, the new salary information and the emp_id is obtained from the object identified by the Name field (e.g. 'emp104'). Also, the last part of the statement (and salary > 3000) is probably useless.

The INSERT statement is built from the following model:

```
INSERT INTO table_from_query_string ([field1, ][field2, ...] list_of_fields)
VALUES ([val1, ][val2, ] ...)
```

Our update example becomes:

```
INSERT INTO employees (emp_id, salary) VALUES ('105', 6500)
```

The INSERT statement is limited to the first table specified in the query string. You can insert records only into real tables, not into views.

Sequential queries

In the current implementation, you are not limited in the number of active sequential queries you have at any time.

Sequential writes are not implemented. You can easily replace a sequential write by an atomic write.

Error Reporting

The Rules Element will report any SQL error message generated by SYBASE in the transcript window (if this window is write enabled). It will also generate error messages if it encounters problems while building the SQL strings. You can consult the SYBASE System Administration Guide for a detailed explanation of the SYBASE messages. Additional error messages are explained in the Open Client DB-Library Reference Manual.

Retrieve Datatype Mapping

The following table indicates how various SYBASE datatypes may (or may not) be retrieved into various Rules Element datatypes. The Rules Element datatypes are listed (underlined> across the top; the SYBASE datatypes are listed in the column to the left. A "Y" means that the operation works with no additional effort or concerns. A number means that the operation is possible, but you should see the notes that appear below the table for additional details. A "--" means that the operation is not possible.

	<u>Integer</u>	<u>Float</u>	<u>Boolean</u>	<u>String</u>	<u>Date</u>
integer (not scaled)	Y	5	4	3	6
integer (scaled)	Y	5	4	3	6
smallint	7	5	4	3	6
quadword	Y	5	4	3	6
tinyint	7	5	4	3	6
real	Y	8	4	3	6
double precision	Y	8	4	3	6
char(n)	Y	Y	Y	Y	Y
varchar(n)	Y	Y	Y	Y	Y
date	6	6	4	2	1

Notes

1. Conversion from an integer value to a float will take place.
2. If the string contains the proper numeric type requested, it will be copied into the Rules Element property.
3. SYBASE puts an "AM" or "PM" stamp on times retrieved from the database, requiring a special Rules Element format be defined in order to retrieve this into a date property. A format that should work is 'AMmm" "*d" "yyyy" "h":"mm*P'. The first wildcard match is for single or double date returns (with one or two blanks). The last "*" is dependent on your version of SYBASE which may or may not return seconds (you could add ':ss' to get them) and thousandths of seconds (which the Rules Element won't accept).
4. If the string contains a valid date, the Rules Element will take it if provided in the default Rules Element date format ('Mmm dd yyyy hh:mm:ss;mm dd yy hh:mm:ss;Mmm dd yyyy;mm dd yy;'). If in some other format, a format may be attached to the property to allow its acceptance (e.g. a format of 'mm"/"dd"/"yy' would accept "12/25/90").
5. Formats may be applied to treat most datatype as booleans. A default property has been defined so that any string of the form "True" or "False" (case-insensitive) will be converted to the appropriate Rules Element boolean. For example, if you have integers that are "0" for "False" and "1" for "True", you could assign a format of 'True;False;1;0;' (which make it print out as True/False, even though it comes in as 1/0). The most obvious candidates to use for booleans are the various strings, the various integers, and "bit".

Write Datatype Mapping

The following table indicates how various SYBASE datatypes may (or may not) be written into from various Rules Element datatypes. The Rules Element datatypes are listed (underlined> across the top; the SYBASE datatypes are listed in the column to the left. A "Y" means that the operation works with no additional effort or concerns. A number means that the operation is possible, but you should see the notes that appear below the table for additional details. A "--" means that the operation is not possible.

The Rules Element SYBASE database interface needs to use the "{I}" syntax for integer database field names in the Name field. This instructs the Rules Element to not treat this as a string, but rather as a numeric field (e.g. integer).

	<u>Integer</u>	<u>Float</u>	<u>Boolean</u>	<u>String</u>	<u>Date</u>
int	Y	1	1,4,5	1,5	1,4,5
smallint	1	1	1,4,5	1,5	1,4,5
tinyint	1	1	1,4,5	1,5	1,4,5
float	1	Y	1,4,5	1,5	1,4,5
char(n)	--	--	Y	Y	Y
varchar(n)	--	--	Y	Y	Y
bit	3	3	1,4,5	1,5	1,4,5
money	2	--	--	1,5	1,4,5
date	--	--	4	1,5	Y
text	--	--	--	--	--
binary(n)	--	--	--	--	--
varbinary(n)	--	--	--	--	--
image	--	--	--	--	--
timestamp	--	--	--	--	--

Notes

1. Datatype conversion, as appropriate and if possible, will take place. For example, a Rules Element integer can be placed into a SYBASE tinyint (8 bits), but it must have a value in the allowed range or SYBASE will generate an error and the entire write operation will fail.
2. There are no "cents" passed in. The integer is treated as an integer number of dollars ("\$").
3. A Rules Element integer or float value of "0" will be "0" in the bit field; any other number will be stored as a "1".
4. Formats must be applied to treat booleans as non-string SYBASE datatypes. For example, you could write into an integer field if you use a boolean format of '1;0;True;False' (which accepts True/False, though prints out as 1/0). The most obvious candidates to use for storing booleans are the various string and integer formats. (Strings will directly receive True/False with the default Rules Element format).

5. Since this SYBASE field needs to be entered without quotes, but the Rules Element, by default will put quotes around non-numeric fields, the "{|}" prefix syntax must be used for the database field name to indicate that this is a numeric-like field and should not have quotes provided by the Rules Element.

Notes

The main difference between SYBASE and the screen captures documented in Appendix A, "Database Integration Examples" are as follows:

1. You must remember to specify SYBASE in the Database Editor window (or in the TKB, @TYPE=SYBASE).
2. In all of the examples where you are going to write to a table, you must specify a BEGIN statement that starts a "named" transaction, and an END statement that, for example, commits the transaction. This syntax is slightly different from the standard examples. For example (e.g. ex01syb.tkb):

```
@BEGIN= "begin transaction write_table";
@END= "commit transaction";
```

3. In all of the examples where you are going to retrieve from a table, the SYBASE interface is generally exactly the same as the standard examples.

Related Topics

Databases

Retrieving from Databases

Writing to Databases

SYLK

SYLK is a standard data format used by several commercial spreadsheet software packages, including Excel on the Macintosh and the IBM-PC. The maximum number of fields which can be contained in a SYLK data file is 10,000.

SYLK

When the SYLK type is specified, the data file is processed as a spreadsheet by the Rules Element. As explained previously, each cell of the spreadsheet containing a value must be named with a unique corresponding slot name obj.prop. In Excel this is done with the Define Name command in the Formula menu.

Example: to modify an existing spreadsheet so that it contains the slot value Expenses.Total, select the cell where you want to put the value and enter the string Expenses.Total in the Define Name dialog. You can repeat the operation for other cells and other slot names. The unnamed cells of the spreadsheet will be ignored by the Rules Element during a Retrieve or Write. The Rules Element may or may not dynamically create new objects when it encounters a named cell (see the Create Object topic for details).

When you create a new SYLK file, the Rules Element automatically names the cells with the corresponding slot names. New cells are created in the

first column of the spreadsheet, but you can modify the layout of the spreadsheet later, provided you keep the correct cell names.

SYLKDB

SYLKDB is used when the Excel spreadsheet file (or a portion of it) is treated as a database. The Excel documentation describes how to select a set of rows and columns (a range) and define it as a database with the Set Database command. In this case, cells are not named individually, but the selected range constitutes a database: rows are records and columns are fields.

This format is more functional than the NXP format for storing structured objects and their slots. You can specify a database name in the Begin (@BEGIN) statement of your query. The Rules Element will search for this database name in the spreadsheet file, and will use the range associated with this database name to locate the records and fields. If you leave the Begin statement empty, the Rules Element will use the word Database as the database name.

You can have several databases in a single spreadsheet file. You can define them with different names in Excel and access them as separate tables from the Rules Element (you must use the Begin statement to identify your database range).

Note: The only database range that you can **create** directly from the Rules Element is "Database"; all other ranges must be pre-defined in the Excel file. Also, when adding records, you must pre-size the range to contain at least one row.

Related Topics

Spreadsheets	Retrieving from Databases
Writing to Databases	Query Language
Begin	

WKS

Description

WKS and WKSDB are used to query and update files which follow the WKS format defined by the Lotus 1-2-3 program on the IBM-PC.

Descriptions of SYLK and SYLKDB hold in the case of WKS and WKSDB. The main points are:

- WKS is a spreadsheet format. The cells must be named in Lotus 1-2-3 to be accessible by the Rules Element.

WKSDB is a Lotus 1-2-3 spreadsheet viewed as a database. You must select a database range in Lotus 1-2-3, and assign a name to it. You must specify the database name in the Begin statement of your transaction.

Note: Transfer of data files between the VAX and PC's (in both directions) should not cause any special problem except in the case of WKS files. In the current version, WKS files created on the VAX (with a RHS Write) can be read on the VAX but not on the PC if they contain numeric data (because of differences in the floating point format),

and files created on the PC cannot be read on the VAX because of RMS file format incompatibilities (you cannot transfer them with Kermit-32 because the records are too long; if you transfer them with the VAXmate PC Server, you create RMS files with unterminated records which cannot be converted properly by the CONVERT VMS utility).

Related Topics

Spreadsheets
Writing to Databases
SYLK

Retrieving from Databases
Query Language
Begin

Write Operator

The `Write` operator is used in rules and methods to write information to a database.

Operands

The `Write` operator takes two operands:

- The first operand is either a quoted string constant or an interpretation evaluating to a string constant specifying the name of the file containing the database to be updated or the login name/password for a DBMS.
- The second operand consists of a series of arguments defining the specific update operation to be performed.

Arguments

The second operand may include the following arguments:

@TYPE	Type of database (creator software and file format)
@BEGIN	Command string for opening transaction
@END	Command string for closing transaction
@QUERY	Command string for updating database
@ERROR	Slot name to trap database error message
@ARGS	Argument list for update command
@ATOMS	List of objects or properties affected
@NAME	Correspondence between objects and records
@FIELDS	List of field names to update
@PROPS	List of properties to update from
@SLOTS	List of slots to update from
@FILL	Create new records or files
@UNKNOWN	Write UNKNOWN values
@CURSOR	Current position for sequential update

Note: It is valid to have an empty second operand. When this occurs, the Rules Element will determine the type of database from the filename extension specified in the first argument, and will default to the SYLK type if no extension is specified. Only simple spreadsheet files can be accessed in this case. This operating mode has been maintained to ensure compatibility with earlier versions of the Rules Element.

When entering a `write` action in the Rule Editor or Method Editor, clicking in the space for the second operand displays the Database Editor dialog box for specifying the update arguments interactively, rather than by explicitly typing them in as listed above.

Effect

The designated information is written to the specified database from the Rules Element knowledge base.

Related Topics

Access String	Left-Hand Side Writes
Access String Specification	Right-Hand Side Writes
Arguments Overview	Order of Sources Writes
Database Editor Window	If Change Writes
Interpretations @V(...)	

Look up the following topics in the Chapter One, “Application Development Features” for information related to the `write` operator.

Rules	Classes
Methods	Properties
Actions	String Constants
Objects	

Write Unknown - (@UNKNOWN)

Usage

The Write Unknown setting is meaningful in all types of transactions. It controls whether or not UNKNOWN values should be written by the transaction.

This setting is specified with the Write Unknown check button in the Database Editor windows. In the text form of the knowledge base, it is saved as:

```
@UNKNOWN=TRUE ;
```

or

```
@UNKNOWN=FALSE ;
```

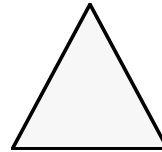
Related Topics

Database Editor Windows	Writing to Databases
Arguments Overview	

Writing to Databases

General

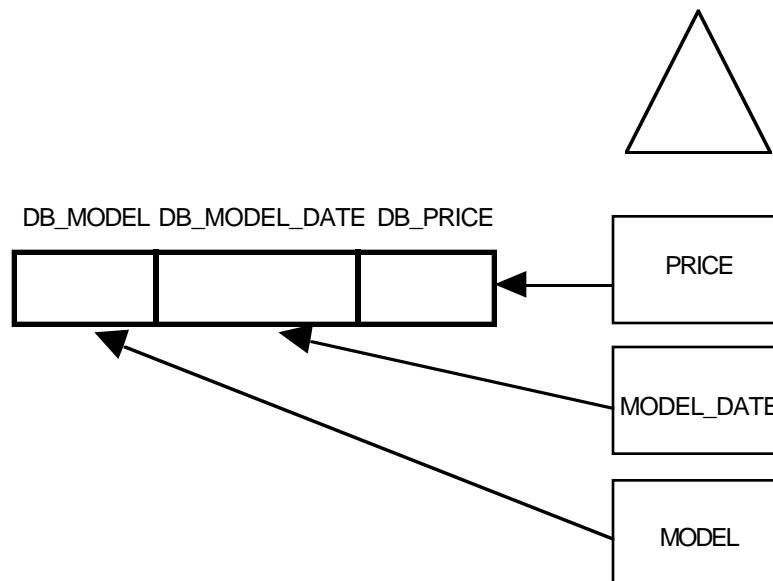
During write operations, the database takes slots and writes them out to the fields in a database record. In most cases, all of the slots are from the same object, thus transforming the Rules Element's object-property relationship into a record-field relationship in the database.



PostScript error (undefined, x6)



For example, take the case of a car inventory file. Each car is represented by a record with the fields `DB_MODEL`, `DB_MODEL_DATE`, and `DB_PRICE`. In the knowledge base, a car is represented by an object with the properties `Model`, `Model_date`, and `Price`. The Write operation in the knowledge base specifies the mapping between the Rules Element properties and the record's fields:



Database Fields	Object Properties
DB_MODEL	Model
DB_MODEL_DATE	Model_Date
DB_PRICE	Price

Each car object's Name, Price, Model, Model_date, and Sportive property slots could be written into a car record's DB_CAR_NAME, DB_PRICE, DB_MODEL, DB_MODEL_DATE, and DB_SPORTIVE fields, respectively. This effectively "transforms" each car object into a car record.

In specialized cases, it's also possible to write the slots from different objects into a record's fields.

Depending on the type of write operation, records can be written one by one from the same slots (with logic in the knowledge base updating the slots before each write), or multiple objects can be written in one operation to many records. During the write, the Rules Element can either update existing records or create new ones.

In most applications, it's not necessary to write all of the slots (object.property combinations) in the Rules Element's working memory to the database. The Rules Element therefore provides several ways of filtering the slots which are actually written from its working memory to the database.

Filtering occurs in several stages:

- A list of slot, object, or class names are provided to the database interface to initially represent the slots to be written. For atomic and sequential writes, the slots are named explicitly; for grouped writes a list of objects or classes is provided from which the slots will be written.
- For grouped writes, existence filtering can be used to determine if a record already exists in the database for the corresponding object. The correlation between a record and an object is established by building a record "key" using the object's name.

Related Topics

Databases	Spreadsheets
Grouped Write	Sequential Write
Atomic Write	Write Operator
Query Write Operations	Slot Specification for Writes
Write Unknown	Create New Record
Debugging Operations	Record Specification for Writes

A

Database Integration Examples

This appendix provides examples of the various ways to use the Intelligent Rules Element database interface.

Example 1 - Grouped Write

Description

In this example data from the slots of two objects is written to the database in a single operation. Each object is written as an individual record. Although this example is oriented towards relational databases, it is also applicable to flat-file databases.

This example uses the following objects and records:

- The class |cars_class| contains two objects: Newcar_1 and Newcar_2. Values have been assigned to their property slots using the InitValue operator in the Order of Sources field.
- Each object has the properties Model, Model_date, Price and Sportive.
- The table CARS contains the columns DB_CAR_NAME, DB_MODEL, DB_MODEL_DATE, DB_PRICE and DB_SPORTIVE.
- There are no rows in the table where the column DB_CAR_NAME contains the value Newcar_1 or Newcar_2.

Operation

Figure A-1 shows the rule that will invoke the grouped write:



Figure A-1 Rule Invoking a Grouped Write

The rule shown above is evaluated as follows:

- The LHS of the rule will always be true.
- The first statement of the RHS (Assign db_access_string...) will prompt the user for the database access string. Look up your database type in Chapter Three, “Database Integration Topics” for details.
- The LoadKB, Assign DropTable, and UnloadKB statements will drop, recreate and reload the CARS table in the database. This will ensure that the CARS table is reset to its initial state each time the example is run.
- A Write operation will be invoked. The argument @V(db_access_string) will be evaluated to yield the user-specified database access string.

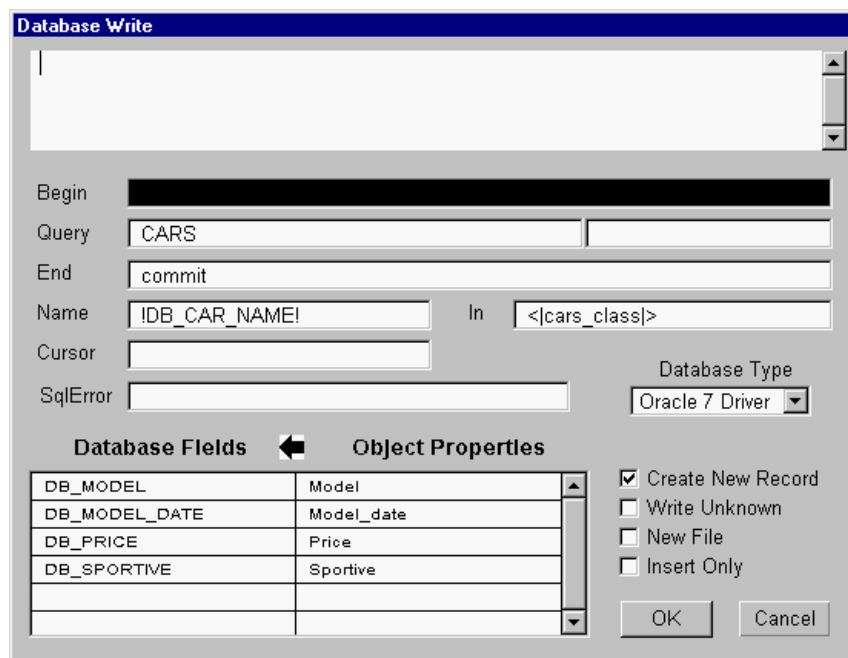


Figure A-2 Write Screen for a Grouped Write

Figure A-2 shows the database interface write screen:

- The object names Newcar_1 and Newcar_2 will be used as keys in the update query.
- The database will try to update those records where the column DB_CAR_NAME contains the values Newcar_1 or Newcar_2. Since there are no rows in the table which satisfy this criteria and Create New Record has been selected, two rows will be inserted into the table CARS using the object names as keys.
- For each of the two objects, the values in property slots Model, Model_date, Price and Sportive will be written to the columns DB_MODEL, DB_MODEL_DATE, DB_PRICE and DB_SPORTIVE, respectively.
- If all of the rows are written successfully, a Commit will be passed to the database.

Reference

Field descriptions for this Write operation follow.

Database Type

An Oracle database is being used in this example.

Begin

For most databases this field is to be left blank for grouped write operations. Some databases, such as Sybase, require a statement here. Look up your database type in Chapter Three, "Database Integration Topics" for details.

Query

This field specifies the database table to which the records are to be written (in this example the table CARS). For flat-file databases this field must be left blank .

End

For Oracle and most other relational databases, this field should contain a Commit statement to make the changes to the table permanent if all rows are written successfully. Look up your database type in Chapter Three, "Database Integration Topics" for details.

Name

This field indicates that the object names are to be used as keys in the DB_CAR_NAME field.

In

Specifying a value of <|class_cars|> indicates that the Rules Element is to write all of the objects in the class |class_cars| to the database.

Cursor

This field must be left blank to indicate a grouped write.

Database Fields / Rules Properties

These columns specify that the values in the property slots Model, Model_date, Price and Sportive are to be written to the columns DB_MODEL, DB_MODEL_DATE, DB_PRICE and DB_SPORTIVE respectively. Although in this example all of the object's property slots are to be written out, this does not necessarily have to be the case.

Create New Record

This is selected to indicate that a new row should be inserted into the table if the update query generated by the Name field fails. Related Topics

Grouped Write

Access String

Query Write Operations

Cursor Slot Specification

Slot Specification for Writes

Database Editor Windows

Also, look up individual arguments and your database type for more detailed information.

Example 2 - Grouped Write with a Complex Name

Description

In this example data from the slots of two objects is written to the database in a single operation. Each object is written as an individual record. Unlike the previous example where the object names could be used in the database as a single-column key to uniquely identify a record, in this example the object names must be parsed into two strings and compared with two database columns in order to determine which records to update. Although this example is oriented towards relational databases, it is also applicable to flat-file databases.

This example uses the following objects and records:

- The class |cars_class| contains two objects: Newcar_1_A_Lexus and Newcar_2_A_Infiniti. Values have been assigned to their property slots using the InitValue operator in the Order of Sources field.
- Each object has the properties Model, Model_date, Price and Sportive.
- The table CARS contains the columns DB_CAR_NAME, DB_MODEL, DB_MODEL_DATE, DB_PRICE and DB_SPORTIVE.
- There are no rows in the table where the columns DB_CAR_NAME and DB_MODEL contain the values car_1 and Lexus, or car_2 and Infiniti.

Operation

Figure A-3 shows the rule which will invoke the grouped write.

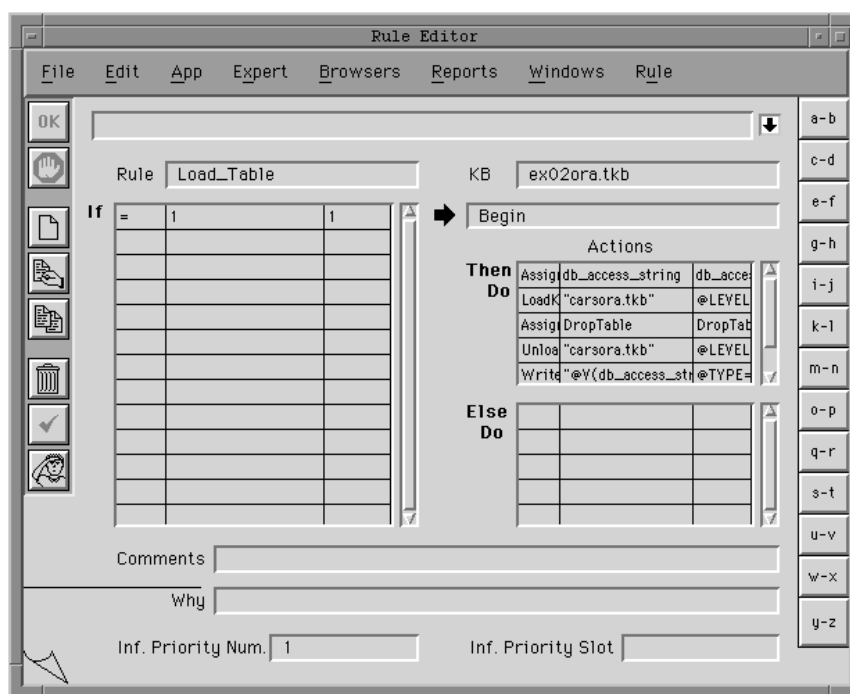


Figure A-3 Rule Invoking a Grouped Write

The rule shown above is evaluated as follows:

- The LHS of the rule will always be true.
- The first statement of the RHS (Assign db_access_string...) will prompt the user for the database access string. Look up your database type for the exact syntax.
- The LoadKB, Assign DropTable, and UnloadKB statements will drop, recreate and reload the CARS table in the database. This will ensure that the CARS table is reset to its initial state each time the example is run.
- A Write operation will be invoked. The argument @V(db_access_string) will be evaluated to yield the user-specified database access string.

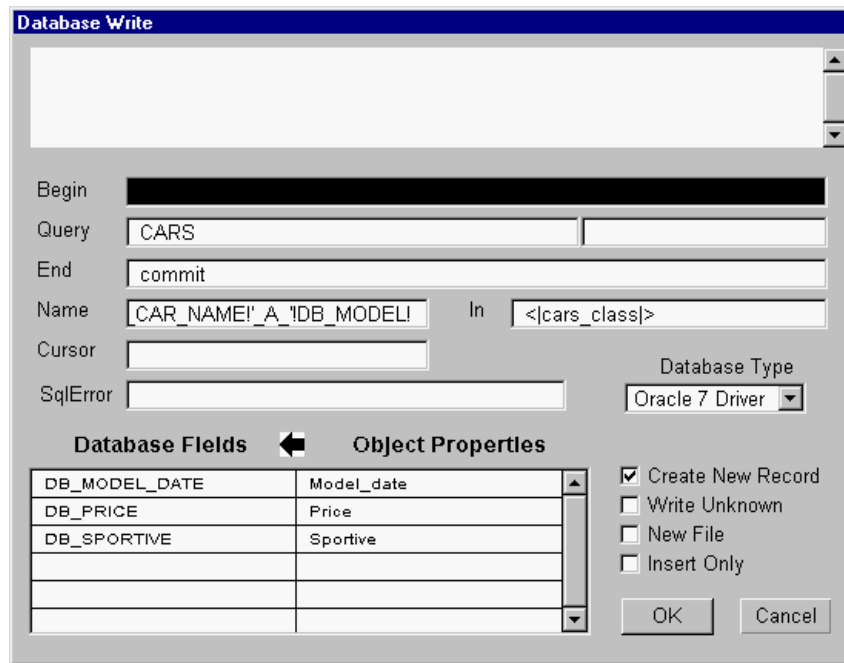


Figure A-4 Write Screen for a Grouped Write Using Name

Figure A-4 shows the Rules Element Write screen:

- The object names will be parsed to yield the values which will be used as keys in the update query. The object Newcar_1_A_Lexus will yield the values car_1 and Lexus, and the object Newcar_2_A_Infiniti will yield the values car_2 and Infiniti.
- The database will try to update those records where the columns DB_CAR_NAME and DB_MODEL contain the values car_1 and Lexus, or car_2 and Infiniti. Since there are no rows in the table which satisfy this criteria and Create New Record has been selected, two rows will be inserted into the table CARS using the parsed values as keys.
- For each of the two objects, the values in property slots Model_date, Price and Sportive will be written to the columns DB_MODEL_DATE, DB_PRICE and DB_SPORTIVE, respectively.
- If all of the rows are written successfully, a Commit will be passed to the database.

Reference

Field descriptions for this Write operation follow.

Database Type

An Oracle database is being used in this example.

Begin

For most databases this field is to be left blank for grouped write operations. Some databases, such as Sybase, require a statement here. Look up your database type for the exact syntax.

Query

This field specifies the database table to which the records are to be written (in this example the table CARS). For flat-file databases this field must be left blank. Look up your database type for the exact syntax.

End

For Oracle and most other relational databases, this field should contain a Commit statement to make the changes to the table permanent if all rows are written successfully. Look up your database type for the exact syntax.

Name

This field indicates how the object names are to be parsed and in which database columns they will be used as keys. In this example, 'New!DB_CAR_NAME!'_A_'!DB_MODEL!' specifies that the write query is to search for records where the column DB_CAR_NAME contains the substring delimited by New and _A_ and where the column DB_MODEL contains the substring which begins after _A_.

In

Specifying a value of <|class_cars|> indicates that the Rules Element is to write all of the objects in the class |class_cars| to the database.

Cursor

This field must be left blank to indicate a grouped write.

Database Fields / Rules Properties

These columns specify that the values in the property slots Model_date, Price and Sportive are to be written to the columns DB_MODEL_DATE, DB_PRICE and DB_SPORTIVE respectively. Although in this example all of the object's property slots are to be written out, this does not necessarily have to be the case.

Create New Record

This is selected to indicate that a new row should be inserted into the table if the update query generated by the Name field fails.

Related Topics

Grouped Write	Writing to Databases
Access String	Slot Specification for Write
Query Write Operations	Database Editor Windows
Record Specification for Writes	

Also, look up individual arguments and your database type for more detailed information.

Example 3 - Atomic Write

Description

In this example one record in a database is updated with the data from the slots of a single object. Although this example is oriented towards relational databases, it is also applicable to flat-file databases.

This example uses the following objects and records:

- The class |cars_class| contains one object: MyCar. Values have been assigned to its property slots using the InitValue operator in the Order of Sources field.
- The object MyCar has the properties Model, Model_date, Price and Sportive.
- The object dummy_object has a single property, dummy_cursor.
- The table CARS contains the columns DB_CAR_NAME, DB_MODEL, DB_MODEL_DATE, DB_PRICE and DB_SPORTIVE.
- The table CARS contains ten records, each of which can be uniquely identified by the values car_1, car_2, car_3,... in the field DB_CAR_NAME.

Operation

Figure A-5 shows the rule which will invoke the atomic write.

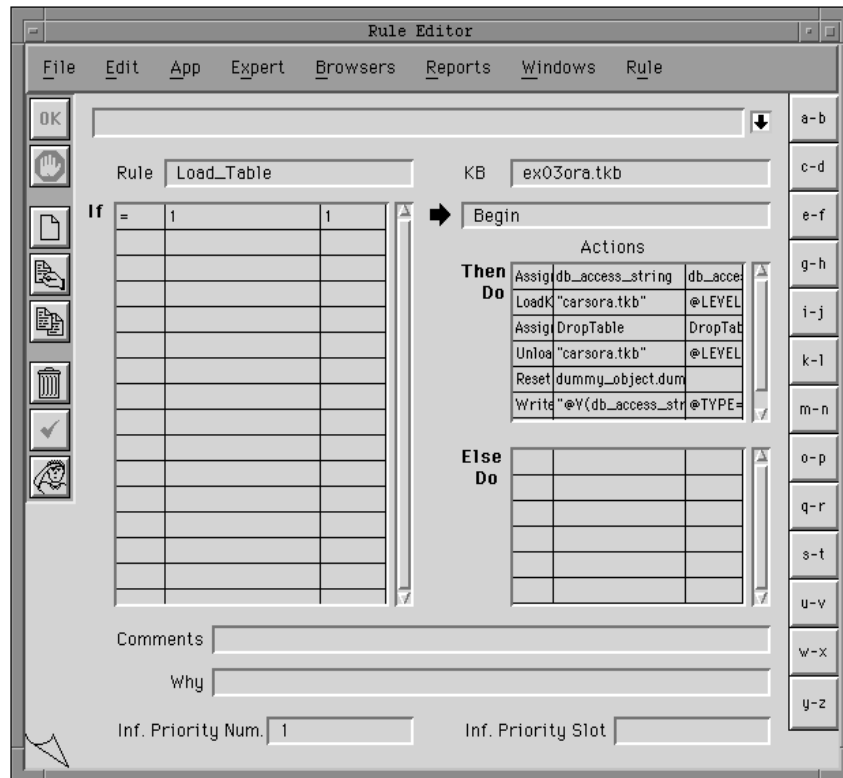


Figure A-5 Rule Invoking an Atomic Write

The rule shown above is evaluated as follows:

- The LHS of the rule will always be true.
- The first statement of the RHS (Assign db_access_string...) will prompt the user for the database access string. Look up your database type in Chapter Three, “Database Integration Topics” for details.
- The LoadKB, Assign DropTable, and UnloadKB statements will drop, recreate and reload the CARS table in the database. This will ensure that the CARS table is reset to its initial state each time the example is run.
- Reset dummy_object.dummy_cursor will set the value of dummy_object.dummy_cursor to UNKNOWN. This will signal the Rules Element that an atomic write will be performed.
- A Write operation will be invoked. The argument @V(db_access_string) will be evaluated to yield the user-specified database access string.

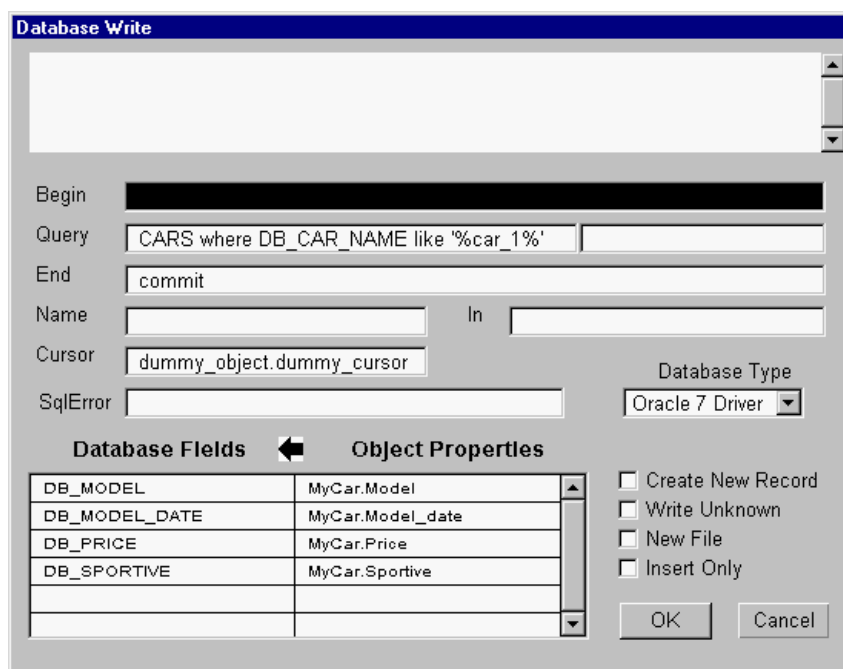


Figure A-6 Write Screen for an Atomic Write

Figure A-6 shows the Rules Element Write screen:

- Data from the slots MyCar.Model, MyCar.Model_date, MyCar.Price and MyCar.Sportive will update the fields DB_MODEL, DB_MODEL_DATE, DB_PRICE and DB_SPORTIVE in the record where DB_CAR_NAME is car_1.
- If the row is written successfully, a Commit will be passed to the database.

Reference

Field descriptions for this Write operation follow.

Database Type

An Oracle database is being used in this example.

Begin

For most databases this field should be left blank for atomic write operations. Some databases, such as Sybase, require a statement here. Look up your database type in Chapter Three, "Database Integration Topics" for details.

Query

This field specifies which database table is to be updated (in this example the table CARS), and the criteria to be used to select the record to be updated (where DB_NAME like...).

End

For Oracle and most other relational databases, this field should contain a Commit statement to make the changes to the table permanent, if the row is updated successfully. Look up your database type in Chapter Three, "Database Integration Topics" for details.

Name

This field must be empty for atomic writes. Object names are stated explicitly in the Database Fields / Rules Properties list.

In

This field must be empty for atomic writes.

Cursor

This field specifies the name of an integer property slot (in this example dummy_object.dummy_cursor) which, in order to specify an atomic write, MUST contain the value UNKNOWN.

Database Fields / Rules Properties

These columns specify that the values in the property slots Model, Model_date, Price and Sportive of the object MyCar are to be written to the columns DB_MODEL, DB_MODEL_DATE, DB_PRICE and DB_SPORTIVE.

Create New Record

This must NOT be selected. New records cannot be added to the database with atomic or sequential writes. Related Topics

Atomic Write	Cursor Slot Specification
Access String	Slot Specification for Writes
Query Write Operations	Database Editor Windows

Also, look up individual arguments and your database type for more detailed information.

Example 4 - Grouped Retrieve

Description

In this example data from multiple records in the database is retrieved into the property slots of a group of objects in a single operation. Although this example is oriented towards relational databases, it is also applicable to flat-file databases.

This example uses the following objects and records:

- Initially, the class |cars_class| contains no objects. Objects in |cars_class| will have the properties Model, Model_date, Price and Sportive.
- The table CARS contains the columns DB_CAR_NAME, DB_MODEL, DB_MODEL_DATE, DB_PRICE and DB_SPORTIVE.
- The table CARS contains ten records, each of which can be uniquely identified by the values car_1, car_2, car_3,... in the field DB_CAR_NAME.

Operation

Figure A-7 shows the rule which will invoke the grouped retrieve.



Figure A-7 Rule Invoking a Grouped Retrieve

The rule shown above is evaluated as follows:

- The LHS of the rule is always true.
- The first statement of the RHS (Assign db_access_string...) will prompt the user for the database access string. Look up your database type in Chapter Three, “Database Integration Topics” for details.
- The LoadKB, Assign DropTable, and UnloadKB statements will drop, recreate and reload the CARS table in the database. This will ensure that the CARS table is reset to its initial state each time the example is run.

- A Retrieve operation will be invoked. The argument @V(db_access_string) will be evaluated to yield the user-specified database access string.

Figure A-8 Retrieve Screen for a Grouped Retrieve

Figure A-8 shows the Rules Element Retrieve screen:

- As each record in the table *CARS* is retrieved, the Rules Element will search the knowledge base for an object whose name matches the current value of the field *DB_CAR_NAME*. Since no object will be found, a dynamic object with this name will be created and linked to the class *|car_class|*.
- As each object is created values from the database fields *DB_MODEL*, *DB_MODEL_DATE*, *DB_PRICE* and *DB_SPORTIVE* will be passed to the property slots *Model*, *Model_date*, *Price* and *Sportive*.

Reference

Field descriptions for this Retrieve operation follow.

Database Type

An Oracle database is being used in this example.

Begin

For most databases this field should be left blank for grouped retrieve operations. Some databases require a statement here. Look up your database type in Chapter Three, "Database Integration Topics" for details.

Query

This field specifies from which table records are to be retrieved. This field can also contain a where clause to limit the records to be retrieved.

End

For most relational databases, this field should be left blank. Some databases may require a statement here. Look up your database type in Chapter Three, “Database Integration Topics” for details.

Name

This field specifies that for each record retrieved from the database the value of the field DB_CAR_NAME is to be used to build the object name in which the database values will be stored.

In

This field specifies the list of objects and/or classes to be searched to determine if an object exists whose name matches the value specified by the Name field. If this field is left blank, as in this example, then all of the objects in the knowledge base will be searched.

Cursor

This field must be empty for grouped retrieves.

Link To

This field specifies the class to which new objects created by the retrieve are to be linked. In this example, new objects will be linked to the class |cars_class|.

Database Fields / Rules Properties

These columns specify that data from the columns DB_MODEL, DB_MODEL_DATE, DB_PRICE and DB_SPORTIVE are to be passed to the property slots Model, Model_date, Price and Sportive .

Create New Record

This is selected to indicate that if an object with a name specified by the Name field doesn't already exist, it is to be created. If this is not selected, data will only be retrieved into objects which already exist in the knowledge base.

Current Forward

This is selected to indicate that the retrieval of any data into property slots will place hypotheses on the agenda according to the current RHS forward-chaining strategy.

Related Topics

Grouped Retrieve

Cursor Slot Specification

Access String

Slot Specification for Retrieves

Object Name Specification

Query Retrieve Operations

Database Editor Windows

Also, look up individual arguments and your database type for more detailed information.

Example 5 - Grouped Retrieve with a Complex Name

Description

This is an example of a grouped retrieve in which field values from two table columns are combined with a constant string to form the object names. Although this example is oriented towards relational databases, it is also applicable to flat-file databases.

This example uses the following objects and records:

- Initially, the class |cars_class| contains no objects. Objects in |cars_class| will have the properties Model, Model_date, Price and Sportive.
- The table CARS contains the columns DB_CAR_NAME, DB_MODEL, DB_MODEL_DATE, DB_PRICE and DB_SPORTIVE.
- The table CARS contains ten records, each of which can be uniquely identified by the values car_1, car_2, car_3,... in the field DB_CAR_NAME.
- The table CARS contains ten records, each of which can be uniquely identified by the values car_1, car_2, car_3,... in the column DB_CAR_NAME. The column DB_MODEL contains values like Toyota, Honda and BMW.

Operation

Figure A-9 shows the rule which will invoke the grouped retrieve.

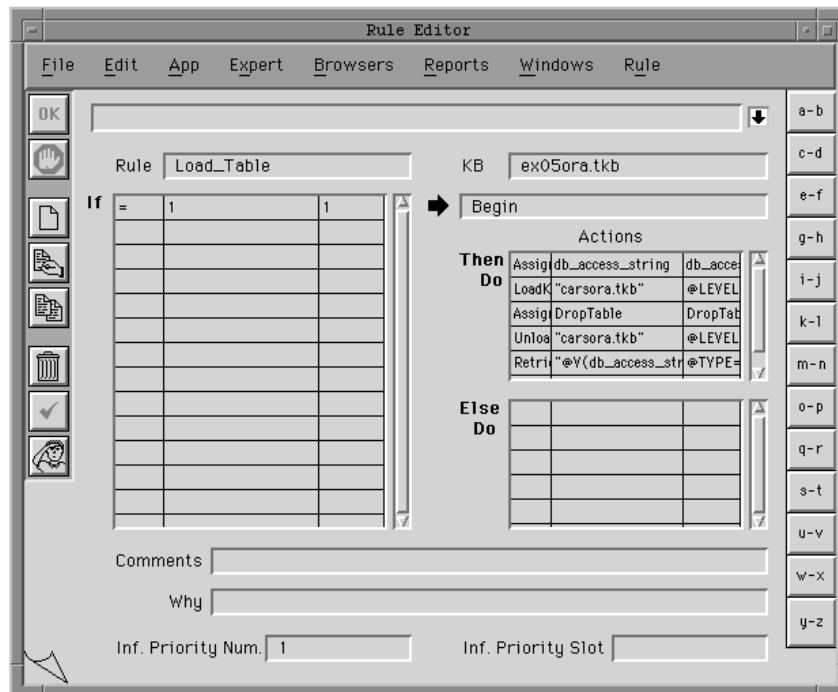


Figure A-9 Rule Invoking a Grouped Retrieve

The rule shown above is evaluated as follows:

- The LHS of the rule is always true.
- The first statement of the RHS (Assign db_access_string...) will prompt the user for the database access string. Look up your database type for the exact syntax.
- The LoadKB, Assign DropTable, and UnloadKB statements will drop, recreate and reload the CARStable in the database. This will ensure that the CARS table is reset to its initial state each time the example is run.
- A Retrieve operation will be invoked. The argument @V(db_access_string) will be evaluated to yield the user-specified database access string.

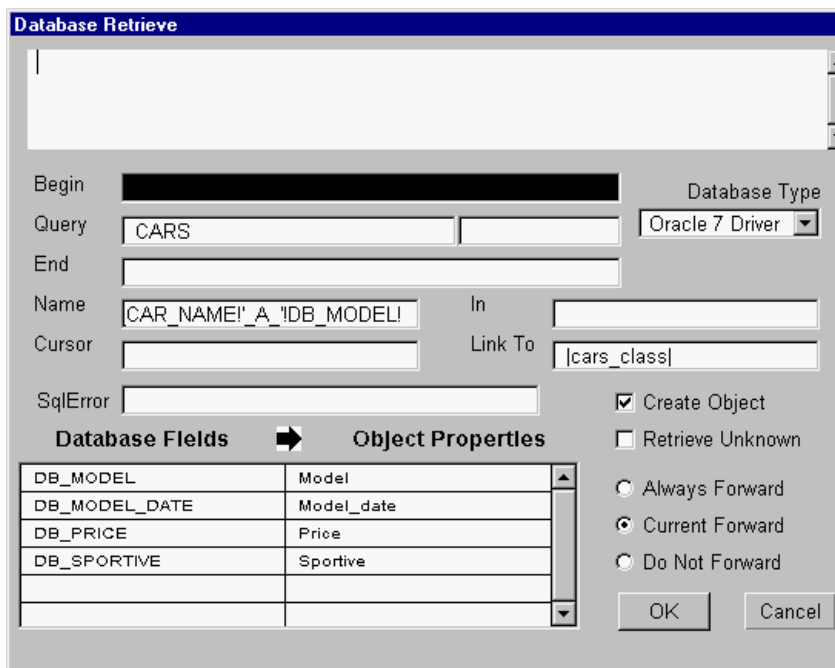


Figure A-10 Retrieve Screen for a Grouped Retrieve Using Name

Figure A-10 shows the Rules Element Retrieve screen:

- As each record in the table CARS is retrieved, the Rules Element will combine the value of the field DB_CAR_NAME with the string _A_ and the value of the field DB_MODEL to create an object name. The Rules Element will then search the knowledge base for an object with this name. Since no object will be found, a dynamic object with this name will be created and linked to the class |car_class|.
- As each object is created values from the database fields DB_MODEL_DATE, DB_PRICE and DB_SPORTIVE will be passed to the property slots Model_date, Price and Sportive.

Reference

Field descriptions for this Retrieve operation follow.

Database Type

An Oracle database is being used in this example.

Begin

For most databases this field should be left blank for grouped retrieve operations. Some databases require a statement here. Look up your database type for the exact syntax.

Query

This field specifies from which table records are to be retrieved. This field can also contain a where clause to limit the records to be retrieved.

End

For most relational databases, this field should be left blank. Some databases may require a statement here. Look up your database type for the exact syntax.

Name

This field specifies that for each record retrieved from the database the value of the field DB_CAR_NAME, the string _A_ and the value of the field DB_MODEL will be combined to form the object name in which the database values will be stored.

In

This field specifies the objects and/or classes of objects to be searched to determine if an object exists whose name matches the value of the database field(s) specified in the Name field. If this field is left blank, as in this example, then all of the objects in the knowledge base will be searched.

Cursor

This field must be empty for grouped retrieves.

Link To

This field specifies the class to which new objects created by the retrieve are to be linked. In this example, new objects will be linked to the class |cars_class|.

Database Fields / Rules Properties

These columns specify that data from the columns DB_MODEL_DATE, DB_PRICE and DB_SPORTIVE are to be passed to the property slots Model_date, Price and Sportive.

Create New Record

This is selected to indicate that if an object with a name specified by the Name field doesn't already exist, it is to be created. If this is not selected, data will only be retrieved into objects which already exist in the knowledge base, and any other records ignored.

Current Forward

This is selected to indicate that the retrieval of any data into property slots will place hypotheses on the agenda according to the RHS forward-chaining strategy currently in effect.

Related Topics

- | | |
|---------------------------|----------------------------------|
| Grouped Retrieve | Cursor Slot Specification |
| Access String | Slot Specification for Retrieves |
| Object Name Specification | Query Retrieve Operations |
| Database Editor Windows | |

Also, look up individual arguments and your database type for more detailed information.

Example 6 - Grouped Retrieve with Existence Filtering

Description

This is an example of a grouped retrieve in which database values are only passed to those objects specified by the In field which already exist in the knowledge base. Although this example is oriented towards relational databases, it is also applicable to flat-file databases.

This example uses the following objects and records:

- The class |cars_class| contains two objects, car_1 and car_2. These objects have the properties Model, Model_date, Price and Sportive. Initially, all of these property slots are set to UNKNOWN for both objects.
- The table CARS contains the columns DB_CAR_NAME, DB_MODEL, DB_MODEL_DATE, DB_PRICE and DB_SPORTIVE.
- The table CARS contains ten records, each of which can be uniquely identified by the values car_1, car_2, car_3,... in the field DB_CAR_NAME.

Operation

Figure A-11 shows the rule which will invoke the grouped retrieve.

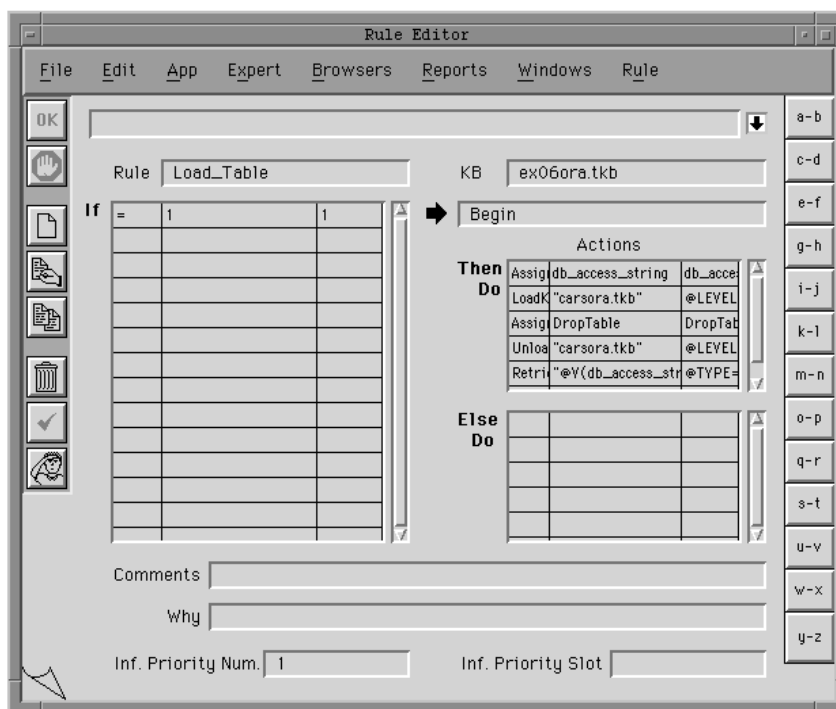


Figure A-11 Rule Invoking a Grouped Retrieve

The rule shown above is evaluated as follows:

- The LHS of the rule is always true.
- The first statement of the RHS (Assign db_access_string...) will prompt the user for the database access string. Look up your database type for details on how to specify this for other DBMSs.
- The LoadKB, Assign DropTable, and UnloadKB statements will drop, recreate and reload the CARStable in the database. This will ensure that the CARS table is reset to its initial state each time the example is run.
- A Retrieve operation will be invoked. The argument @V(db_access_string) will be evaluated to yield the user-specified database access string.

The screenshot shows a 'Database Retrieve' dialog box with the following fields and values:

- Begin: [Redacted]
- Query: CARS
- End: [Empty]
- Name: IDB_CAR_NAME!
- Cursor: [Empty]
- Database Type: Oracle 7 Driver
- In: <|cars_class|>
- Link To: [Empty]
- SqlError: [Empty]
- Object Properties:

Database Fields	Object Properties
DB_MODEL	Model
DB_MODEL_DATE	Model_date
DB_PRICE	Price
DB_SPORTIVE	Sportive
- Radio buttons: Always Forward, **Current Forward**, Do Not Forward
- Checkboxes: Create Object, Retrieve Unknown
- Buttons: OK, Cancel

Figure A-12 Retrieve Screen for a Grouped Retrieve Using In Field

Figure A-12 shows the Rules Element retrieve screen:

- As each record in the table CARS is retrieved, the Rules Element will search the objects in the class <|cars_class|> (as specified by the In field) for an object whose name matches the current value of the field DB_CAR_NAME.
- Only two records will have values in the field DB_CAR_NAME which match the name of an object in the class <|cars_class|>.
- For the two objects car_1 and car_2, the values from the database fields DB_MODEL, DB_MODEL_DATE, DB_PRICE and DB_SPORTIVE will be passed to the property slots Model, Model_date, Price and Sportive. Data from other records retrieved will be ignored.

Reference

Field descriptions for this Retrieve operation follow.

Database Type

An Oracle database is being used in this example.

Begin

For most databases this field should be left blank for grouped retrieve operations. Some databases require a statement here. Look up your database type for details.

Query

This field specifies from which table records are to be retrieved. This field can also contain a where clause to limit the records to be retrieved.

End

For most relational databases, this field should be left blank. Some databases may require a statement here. Look up your database type for the exact syntax for your database.

Name

This field specifies that for each record retrieved, the Rules Element is to search for objects whose name matches the value of the field DB_CAR_NAME.

In

This field specifies the list of objects and/or classes of objects to be searched to determine if an object exists whose name matches the value specified by the Name field. In this example, data will only be passed to existing objects in the class <| cars_class |>.

Cursor

This field must be empty for grouped retrieves.

Link To

Since no objects are to be created by this retrieve, this field is left empty.

Database Fields / Rules Properties

These columns specify that data from the columns DB_MODEL, DB_MODEL_DATE, DB_PRICE and DB_SPORTIVE are to be passed to the property slots Model, Model_date, Price and Sportive .

Create New Record

Since this is not selected, data will only be retrieved into objects which already exist in the knowledge base. Any other records will be ignored.

Current Forward

This is selected to indicate that the retrieval of any data into property slots will place hypotheses on the agenda according to the current RHS forward-chaining strategy.

Related Topics

Grouped Retrieve
Access String

Cursor Slot Specification
Slot Specification for Retrieves

Object Name Specification Query Retrieve Operations
 Database Editor Windows Existence Filtering Operations

Also, look up individual arguments and your database type for more detailed information.

Example 7 - Grouped Retrieve with Content Filtering

Description

This is an example of a grouped retrieve in which the records retrieved are limited by a database query. Although this example is oriented towards relational databases, it is also applicable to flat-file databases.

This example uses the following objects and records:

- Initially, the class |cars_class| contains no objects. Objects in |cars_class| will have the properties Model, Model_date, Price and Sportive.
- The table CARS contains the columns DB_CAR_NAME, DB_MODEL, DB_MODEL_DATE, DB_PRICE and DB_SPORTIVE.
- The table CARS contains ten records, each of which can be uniquely identified by the values car_1, car_2, car_3,... in the field DB_CAR_NAME. Five records have a Sportive field with a value of Yes: car_1, car_4, car_5, car_7 and car_8.

Operation

Figure A-13 shows the rule which will invoke the grouped retrieve.

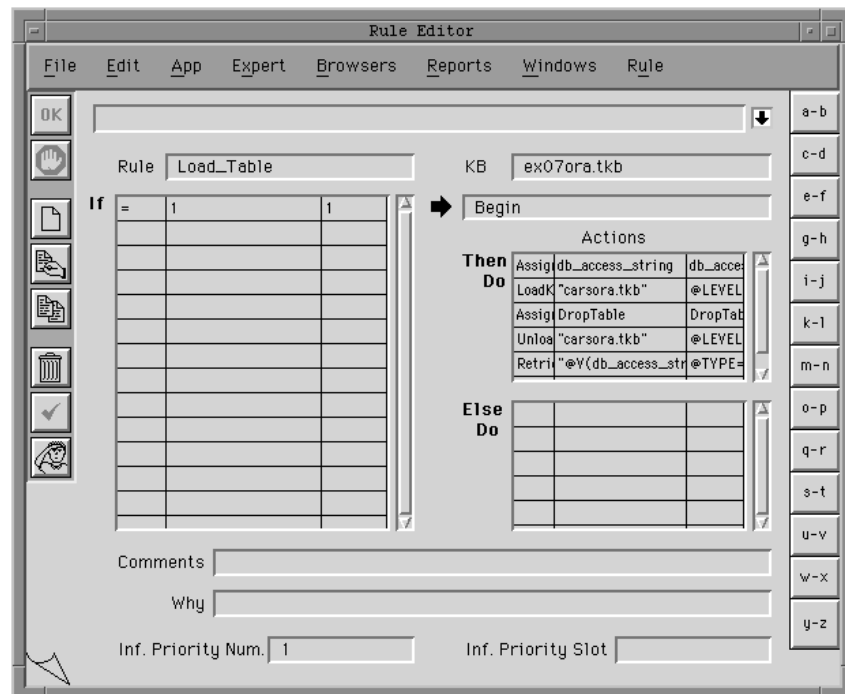


Figure A-13 Rule Invoking a Grouped Retrieve

The rule shown above is evaluated as follows:

- The LHS of the rule is always true.
- The first statement of the RHS (Assign db_access_string...) will prompt the user for the database access string. Look up your database type for information on how to specify this for other DBMSs.
- The LoadKB, Assign DropTable, and UnloadKB statements will drop, recreate and reload the CARStable in the database. This will ensure that the CARS table is reset to its initial state each time the example is run.
- A Retrieve operation will be invoked. The argument @V(db_access_string) will be evaluated to yield the user-specified database access string.

Database Retrieve

Begin [Redacted] Database Type Oracle 7 Driver

Query CARS where DB_SPORTIVE='Yes'

End [Redacted]

Name [Redacted] in [Redacted]

Cursor |DB_CAR_NAME| Link To |cars_class|

SqlError [Redacted]

Create Object

Retrieve Unknown

Always Forward

Current Forward

Do Not Forward

Database Fields	Object Properties
DB_MODEL	Model
DB_MODEL_DATE	Model_date
DB_PRICE	Price
DB_SPORTIVE	Sportive

OK Cancel

Figure A-14 Retrieve Screen for a Grouped Retrieve Using Link To

Figure A-14 shows the Rules Element retrieve screen:

- Since the SQL query CARS where DB_SPORTIVE = 'Yes' has been specified, the DBMS will return only those records which satisfy this condition.
- As each record in the table CARS is retrieved, the Rules Element will search the knowledge base for an object whose name matches the current value of the field DB_CAR_NAME. Since no object will be found, a dynamic object with this name will be created and linked to the class |car_class|.
- As each object is created values from the database fields DB_MODEL, DB_MODEL_DATE, DB_PRICE and DB_SPORTIVE will be passed to the property slots Model, Model_date, Price and Sportive.

Reference

Field descriptions for this Retrieve operation follow.

Database Type

An Oracle database is being used in this example.

Begin

For most databases this field should be left blank for grouped retrieve operations. Some databases require a statement here. Look up your database type for details.

Query

This field specifies from which table records are to be retrieved and the criteria to be used to select the desired records. In this example, only records which have a Yes value in the field DB_SPORTIVE will be retrieved.

End

For most relational databases, this field should be left blank. Some databases may require a statement here. Look up your database type for the exact syntax for your database.

Name

This field specifies that for each record retrieved from the database the value of the field DB_CAR_NAME is to be used to build the object name in which the database values will be stored.

In

This field specifies the list of objects and/or classes to be searched to determine if an object exists whose name matches the value specified by the Name field. If this field is left blank, as in this example, then all of the objects in the knowledge base will be searched.

Cursor

This field must be empty for grouped retrieves.

Link To

This field specifies the class to which new objects created by the retrieve are to be linked. In this example, new objects will be linked to the class |cars_class|.

Database Fields / Rules Properties

These columns specify that data from the columns DB_MODEL, DB_MODEL_DATE, DB_PRICE and DB_SPORTIVE are to be passed to the property slots Model, Model_date, Price and Sportive .

Create New Record

This is selected to indicate that if an object with a name specified by the Name field doesn't already exist, it is to be created. If this is not selected, data will only be retrieved into objects which already exist in the knowledge base.

Current Forward

This is selected to indicate that the retrieval of any data into property slots will place hypotheses on the agenda according to the current RHS forward-chaining strategy.

Related Topics

- Grouped Retrieve
- Database Editor Windows
- Object Name Specification
- Query Retrieve Operations
- Slot Specification for Retrieves

Also, look up individual arguments and your database type for more detailed information.

Example 8 - Atomic Retrieve

Description

In this example the property slots of a single object are passed values from a single database record. Although this example is oriented towards relational databases, it is also applicable to flat-file databases.

This example uses the following objects and records:

- The class |cars_class| contains one object: MyCar. It has the properties Model, Model_date, Price and Sportive.
- The object dummy_object has a single property, dummy_cursor.
- The table CARS contains the columns DB_CAR_NAME, DB_MODEL, DB_MODEL_DATE, DB_PRICE and DB_SPORTIVE.
- The table CARS contains ten records, each of which can be uniquely identified by the values car_1, car_2, car_3,... in the field DB_CAR_NAME.

Operation

Figure A-15 shows the rule which will invoke the atomic retrieve

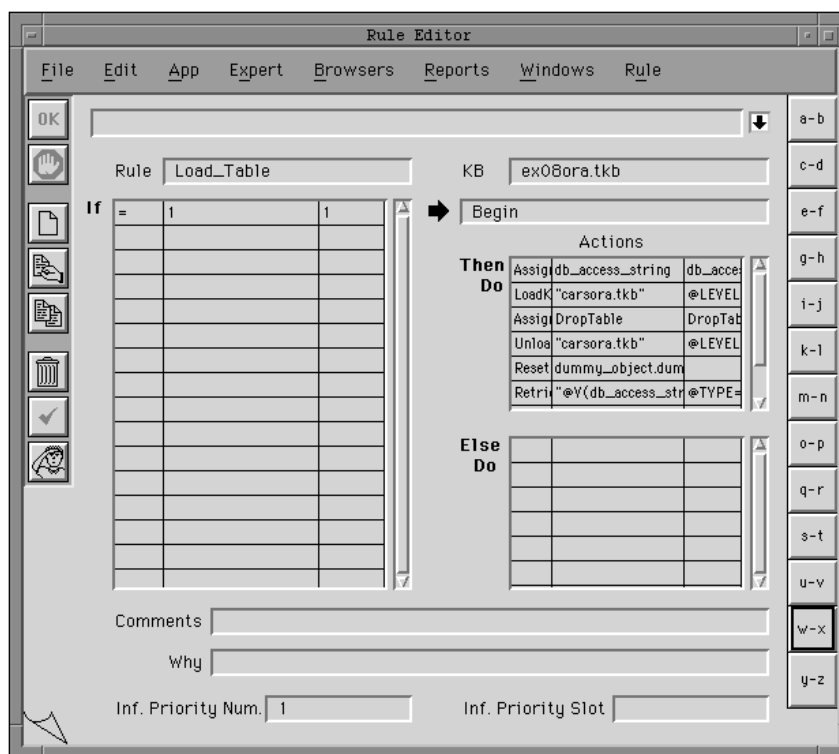


Figure A-15 Rule Invoking an Atomic Retrieve

The rule shown above is evaluated as follows:

- The LHS of the rule is always true.
- The first statement of the RHS (Assign db_access_string...) will prompt the user for the database access string. Look up your database type for information on how to specify this for other DBMSs.
- The LoadKB, Assign DropTable, and UnloadKB statements will drop, recreate and reload the CARS table in the database. This will ensure that the CARS table is reset to its initial state each time the example is run.
- Reset dummy_object.dummy_cursor will set the value of dummy_object.dummy_cursor to UNKNOWN. This will signal the Rules Element that an atomic retrieve will be performed.
- A Retrieve operation will be invoked. The argument @V(db_access_string) will be evaluated to yield the user-specified database access string

Database Fields	Object Properties
DB_MODEL	MyCar.Model
DB_MODEL_DATE	MyCar.Model_date
DB_PRICE	MyCar.Price
DB_SPORTIVE	MyCar.Sportive

Figure A-16 Retrieve Screen for an Atomic Retrieve

Figure A-16 shows the Rules Element retrieve screen:

- Since the SQL query CARS where DB_CAR_NAME = 'car_1' has been specified, the DBMS will return the record which satisfies this condition.
- The values from the database fields DB_MODEL, DB_MODEL_DATE, DB_PRICE and DB_SPORTIVE will be passed to the property slots Model, Model_date, Price and Sportive of the object MyCar.

Reference

Field descriptions for this Retrieve operation follow.

Database Type

An Oracle database is being used in this example.

Begin

For most databases this field should be left blank for atomic retrieve operations. Some databases require a statement here. Look up your database type for details.

Query

This field specifies from which table records are to be retrieved and the criteria to be used to select the desired records. In this example, only the record which has the value `car_1` in the field `DB_CAR_NAME` will be retrieved. If, for an atomic retrieve, the query specified returns more than one record, only the first one will be used; all of the others will be ignored.

End

For most relational databases, this field should be left blank. Some databases may require a statement here. Look up your database type for details.

Name

This field must be empty for atomic retrieves. Object names are stated explicitly in the Database Fields / Rules Properties list.

In

This field is left empty.

Cursor

This field specifies the name of an integer property slot (in this example `dummy_object.dummy_cursor`) which, in order to specify an atomic retrieve, **MUST** contain the value `UNKNOWN`. Upon successful completion of the retrieve, the cursor will be set to 1. It must be reset to `UNKNOWN`, before another atomic retrieve can be performed.

Link To

This field is left empty.

Database Fields / Rules Properties

These columns specify that data from the columns `DB_MODEL`, `DB_MODEL_DATE`, `DB_PRICE` and `DB_SPORTIVE` are to be passed to the property slots `Model`, `Model_date`, `Price` and `Sportive` of the object `MyCar`.

Current Forward

This is selected to indicate that the retrieval of any data into property slots will place hypotheses on the agenda according to the current RHS forward-chaining strategy.

Related Topics

Atomic Retrieve

Database Editor Windows

Cursor Slot Specification

Slot Specification for Retrieves

Object Name Specification Query Retrieve Operations
Retrieving from Databases

Also, look up individual arguments and your database type for more detailed information.

Example 9 - Sequential Retrieve

Description

In this example data from multiple database records is passed to the property slots of a single object one record at a time. The retrieve is invoked once for each record in the table. Although this example is oriented towards relational databases, it is also applicable to flat-file databases.

This example uses the following objects and records:

- The class |cars_class| contains one object: MyCar. It has the properties Model, Model_date, Price and Sportive.
- The object dummy_object has a single property, dummy_cursor.
- The table CARS contains the columns DB_CAR_NAME, DB_MODEL, DB_MODEL_DATE, DB_PRICE and DB_SPORTIVE.
- The table CARS contains ten records, each of which can be uniquely identified by the values car_1, car_2, car_3,... in the field DB_CAR_NAME.Operation

Figure A-17 shows the rule which will invoke the sequential retrieve.

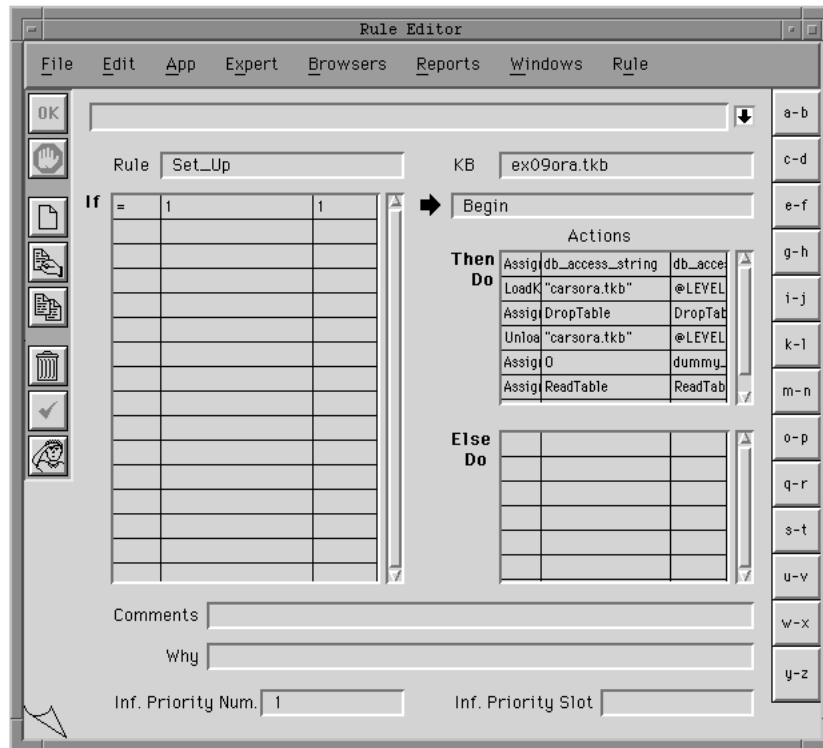


Figure A-17 Rule Initializing a Sequential Retrieve

The rule shown above is evaluated as follows:

- The LHS of the rule is always true.
- The first statement of the RHS (Assign db_access_string...) will prompt the user for the database access string. Look up your database type in Chapter Three, "Database Integration Topics" for details.
- The LoadKB, Assign DropTable, and UnloadKB statements will drop, recreate and reload the CARStable in the database. This will ensure that the CARStable is reset to its initial state each time the example is run.
- Assign dummy_object.dummy_cursor 0 will set the value of dummy_object.dummy_cursor to zero. This will signal the Rules Element that a sequential retrieve will be performed.
- The "Assign ReadTable ReadTable" will invoke the rule which will perform the sequential retrieve.

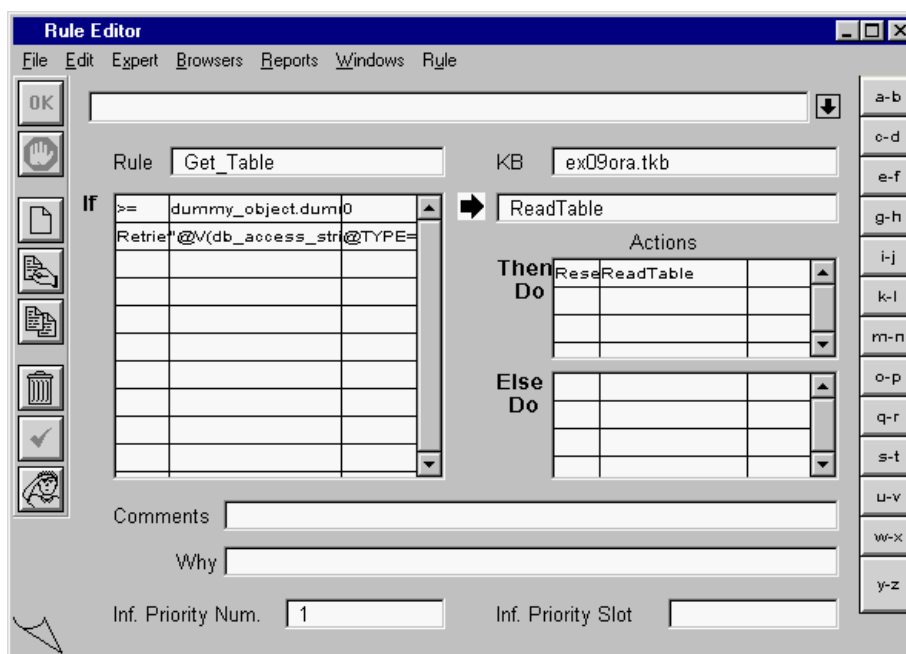


Figure A-18 Rule Invoking a Sequential Retrieve

FigureA-18 shows the rule which will invoke the sequential retrieve. It is evaluated as follows:

- The LHS tests to see if the value of dummy_object.dummy_cursor is greater than or equal to 0. This will be true until the Retrieve fetches the last record, at which point it will be set to -1. At that point, the test will fail and execution will end.
- The second statement of the LHS will invoke the retrieve.
- Reset ReadTable will cause this rule to be re-executed. This, in turn, will re-test the cursor's value, and re-execute the Retrieve until all records have been retrieved.

Note that each time the Retrieve is invoked, it overlays the property slots with the data from the current record. In a real knowledge base, there would undoubtedly be some intermediate processing of the slots before the hypothesis ReadTable is reset and the next retrieve is issued.

Database Fields	Object Properties
DB_MODEL	MyCar.Model
DB_MODEL_DATE	MyCar.Model_date
DB_PRICE	MyCar.Price
DB_SPORTIVE	MyCar.Sportive

Figure A-19 Retrieve Screen for a Sequential Retrieve

Figure A-19 shows the Rules Element retrieve screen:

- The DBMS will retrieve all of the records in the table CARS one record at a time.
- Each time the retrieve is invoked, the values from the database fields DB_MODEL, DB_MODEL_DATE, DB_PRICE and DB_SPORTIVE of the current record will be passed to the property slots Model, Model_date, Price and Sportive of the object MyCar.

Reference

Field descriptions for this Retrieve operation follow.

Database Type

An Oracle database is being used in this example.

Begin

For most databases this field should be left blank for sequential retrieve operations. Some databases require a statement here. Look up your database type in Chapter Three, “Database Integration Topics” for details.

Query

This field specifies from which table records are to be retrieved and the criteria to be used to select the desired records. In this example, all of the records in the table CARS will be retrieved.

End

For most other relational databases, this field should be left blank. Some databases may require a statement here. Look up your database type in Chapter Three, "Database Integration Topics" for details.

Name

This field must be empty for sequential retrieves. Object names are stated explicitly in the Database Fields / Rules Properties list.

In

This field is left empty.

Cursor

This field specifies the name of an integer property slot (in this example `dummy_object.dummy_cursor`) which, in order to specify a sequential retrieve, **MUST** contain the value 0 before the retrieve is invoked for the first time. Each time a record is successfully retrieved, the cursor will be set to 1. When all records have been retrieved, or, if an error has occurred, the cursor will be set to -1.

Link To

This field is left empty.

Database Fields / Rules Properties

These columns specify that data from the columns `DB_MODEL`, `DB_MODEL_DATE`, `DB_PRICE` and `DB_SPORTIVE` are to be passed to the property slots `Model`, `Model_date`, `Price` and `Sportive` of the object `MyCar`.

Current Forward

This is selected to indicate that the retrieval of any data into property slots will place hypotheses on the agenda according to the current RHS forward-chaining strategy.

Related Topics

Sequential Retrieve

Cursor Slot Specification

Access String

Slot Specification for Retrieves

Object Name Specification

Query Retrieve Operations

Database Editor Windows

Also, look up individual arguments and your database type for more detailed information.

Example 10 - Sequential Retrieve with a Parameterized Query

Description

In this example data from multiple database records is passed to the property slots of a single object one record at a time. The retrieve is invoked once for each record in the table. Unlike the previous example, this retrieve employs a query which contains slot values as parameters. Although this

example is oriented towards relational databases, it is also applicable to flat-file databases.

This example uses the following objects and records:

- The class |cars_class| contains one object: MyCar. It has the properties Model, Model_date, Price and Sportive.
- The object dummy_object has a single property, dummy_cursor.
- The object ref_object has two properties, ref_price and ref_sportive. The values 30000 and Yes have been assigned to the property slots using the InitValue operator in the Order of Sources field.
- The table CARS contains the columns DB_CAR_NAME, DB_MODEL, DB_MODEL_DATE, DB_PRICE and DB_SPORTIVE.
- The table CARS contains ten records, each of which can be uniquely identified by the values car_1, car_2, car_3,... in the field DB_CAR_NAME.

Operation

Figure A- 20 shows the rule which will invoke the sequential retrieve.

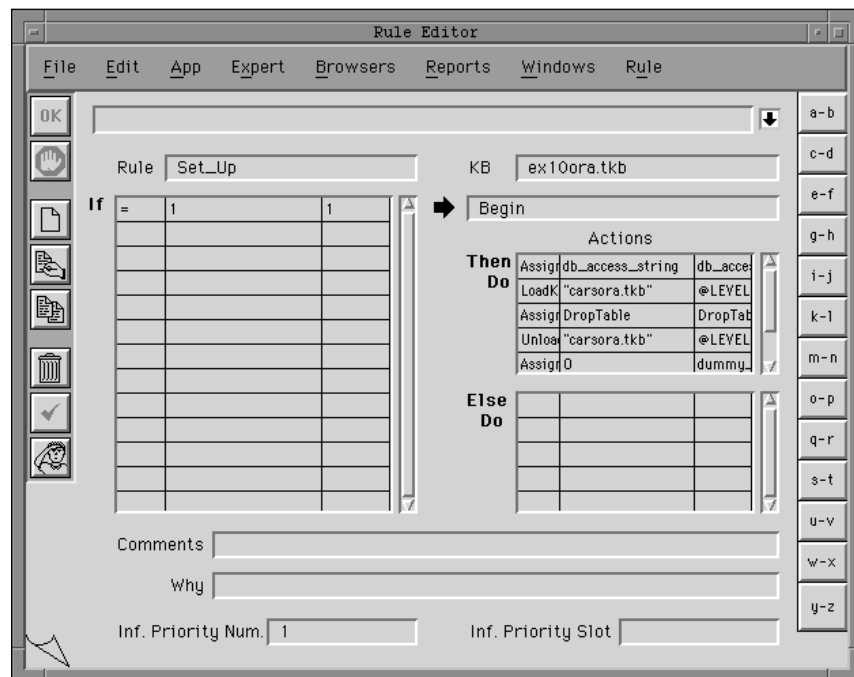


Figure A-20 Rule Initializing a Parameterized Sequential Retrieve

The rule shown above is evaluated as follows:

- The LHS of the rule is always true.
- The first statement of the RHS (Assign db_access_string...) will prompt the user for the database access string. Look up your database type in Chapter Three, "Database Integration Topics" for details.

- The LoadKB, Assign DropTable, and UnloadKB statements will drop, recreate and reload the CARStable in the database. This will ensure that the CARStable is reset to its initial state each time the example is run.
- Assign dummy_object.dummy_cursor 0 will set the value of dummy_object.dummy_cursor to zero. This will signal the Rules Element that a sequential retrieve will be performed.
- The "Assign ReadTable ReadTable" will invoke the rule which will perform the sequential retrieve.

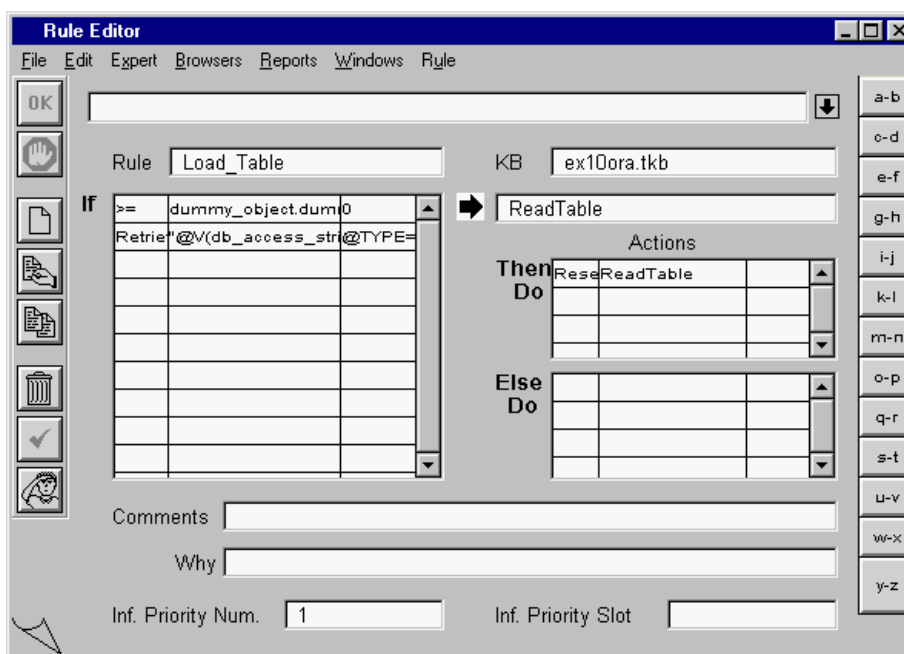


Figure A-21 Rule Invoking a Parameterized Sequential Retrieve

Figure A-21 shows the rule which will invoke the parameterized sequential retrieve. It is evaluated as follows:

- The LHS tests to see if the value of dummy_object.dummy_cursor is greater than or equal to 0. This will be true until the Retrieve fetches the last record, at which point it will be set to -1. At that point, the test will fail and execution will end.
- The second statement of the LHS will invoke the retrieve.
- Reset ReadTable will cause this rule to be re-executed. This, in turn, will re-test the cursor's value, and re-execute the Retrieve until all records have been retrieved.

Note that each time the Retrieve is invoked, it overlays the property slots with the data from the current record. In a real knowledge base, there would undoubtedly be some intermediate processing of the slots before the hypothesis ReadTable is reset and the next Retrieve is issued

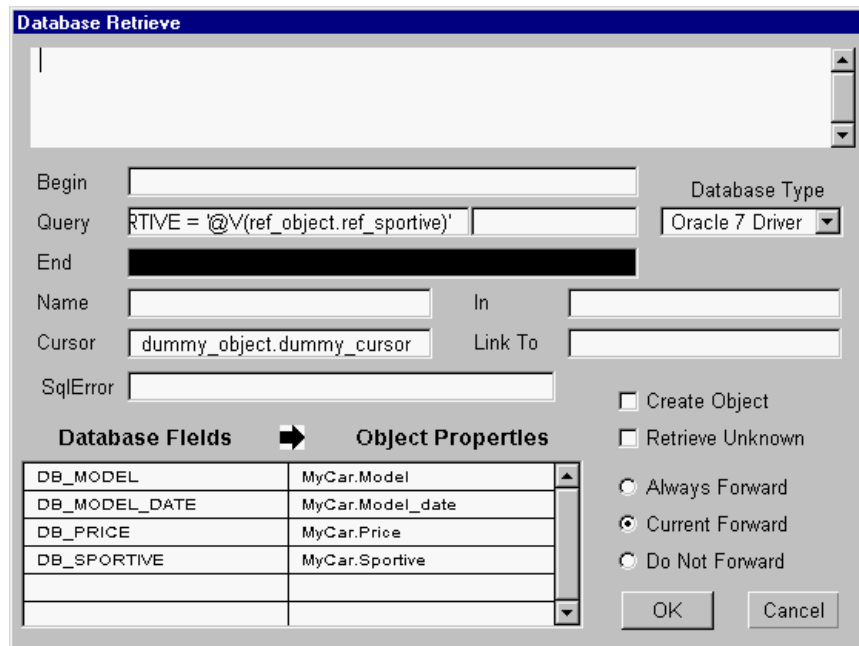


Figure A-22 Retrieve Screen for a Sequential Retrieve Using Query

Figure A-22 shows the Rules Element retrieve screen:

- The variables @V(ref_object.ref_price) and @V(ref_object.ref_sportive) in the query will be replaced by the value of the slots 30000 and Yes respectively. The four records in the table CARS which satisfy this query will be passed to the object MyCar one record at a time.
- Each time the retrieve is invoked, the values from the database fields DB_MODEL, DB_MODEL_DATE, DB_PRICE and DB_SPORTIVE of the current record will be passed to the property slots Model, Model_date, Price and Sportive of the object MyCar.

Reference

Field descriptions for this Retrieve operation follow.

Database Type

An Oracle database is being used in this example.

Begin

For most databases this field should be left blank for sequential retrieve operations. Some databases require a statement here. Look up your database type in Chapter Three, "Database Integration Topics" for details.

Query

This field specifies from which table records are to be retrieved and the criteria to be used to select the desired records. Slot values can be used as query parameters; they can be specified in the query as @v(object.property). Note that the interpretation must be placed in single quotes if it has a value of type string. See the Query field of Figure A-22 for examples.

End

For most other relational databases, this field should be left blank. Some databases may require a statement here. Look up your database type in Chapter Three, "Database Integration Topics" for details.

Name

This field must be empty for sequential retrieves. Object names are stated explicitly in the Database Fields / Rules Properties list.

In

This field is left empty.

Cursor

This field specifies the name of an integer property slot (in this example dummy_object.dummy_cursor) which, in order to specify a sequential retrieve, MUST contain the value 0 before the retrieve is invoked for the first time. Each time a record is successfully retrieved, the cursor will be set to 1. When all records have been retrieved, or, if an error has occurred, the cursor will be set to -1.

Link To

This field is left empty.

Database Fields / Rules Properties

These columns specify that data from the columns DB_MODEL, DB_MODEL_DATE, DB_PRICE and DB_SPORTIVE are to be passed to the property slots Model, Model_date, Price and Sportive of the object MyCar.

Current Forward

This is selected to indicate that the retrieval of any data into property slots will place hypotheses on the agenda according to the current RHS forward-chaining strategy.

Related Topics

Sequential Retrieve	Cursor Slot Specification
Access String	Slot Specification for Retrieves
Object Name Specification	Query Retrieve Operations
Database Editor Windows	Retrieving from Databases

Also, look up individual arguments and your database type for more detailed information.

Example 11 - Grouped Retrieve with a SQL Join

Description

This is an example of a grouped retrieve in which records are retrieved from more than one database table use an SQL join query. Since a join is a concept which only applies to relational databases, this example is not applicable to flat-file databases.

This example uses the following objects and records:

- Initially, the class |cars_class| contains no objects. Objects in |cars_class| will have the properties Model, Model_date, Price, Sportive and Dealer_name.
- The table CARS contains the columns DB_CAR_NAME, DB_MODEL, DB_MODEL_DATE, DB_PRICE and DB_SPORTIVE.
- The table CARS contains ten records, each of which can be uniquely identified by the values car_1, car_2, car_3,... in the field DB_CAR_NAME.
- The table DEALERS contains the columns DB_DEALER_NAME and DB_DEALER_MODEL.
- The table DEALERS contains eight records which relate dealer names and models.

Operation

Figure A-23 shows the rule which will invoke the grouped retrieve

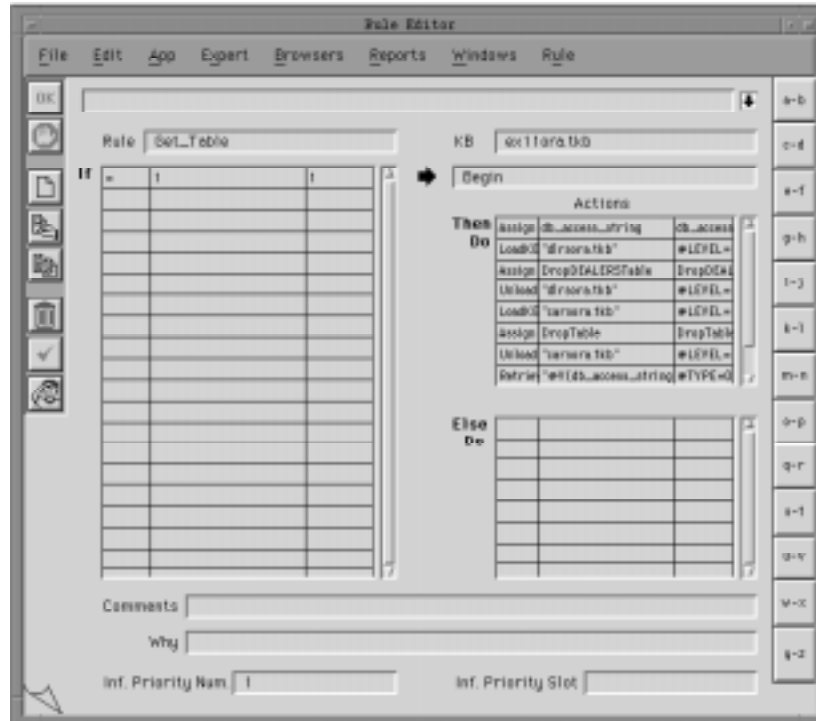


Figure A-23 Rule Invoking a Grouped Retrieve

The rule shown above is evaluated as follows:

- The LHS of the rule is always true.
- The first statement of the RHS (Assign db_access_string...) will prompt the user for the database access string. Look up your database type in Chapter Three, “Database Integration Topics” for details.
- The LoadKB, Assign DropDEALERSTable, and UnloadKB statements will drop, recreate and reload the DEALERS table in the database. This will ensure that the DEALERS table is reset to its initial state each time the example is run.
- The LoadKB, Assign DropTable, and UnloadKB statements will drop, recreate and reload the CARS table in the database. This will ensure that the CARS table is reset to its initial state each time the example is run.
- A Retrieve operation will be invoked. The argument @V(db_access_string) will be evaluated to yield the user-specified database access string

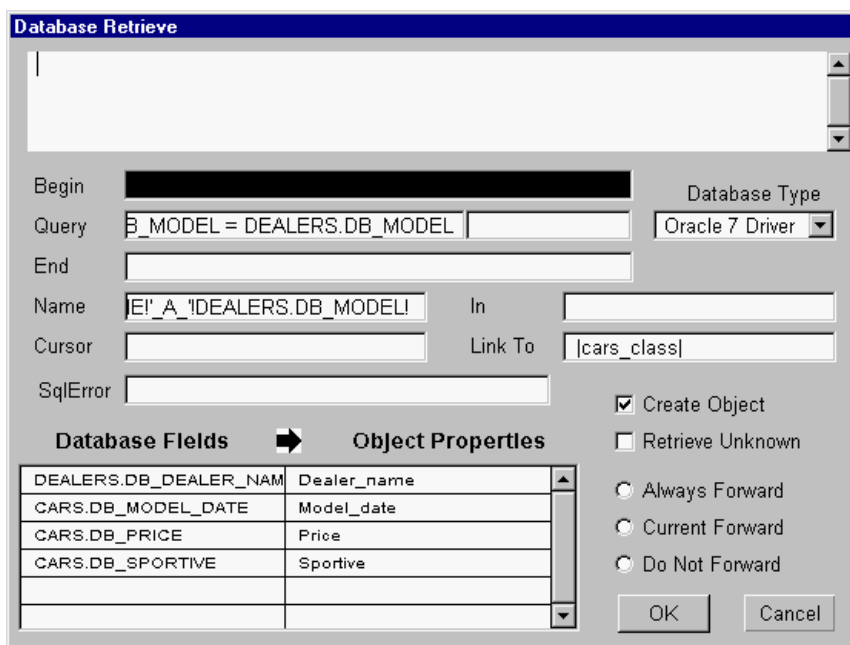


Figure A-24 Retrieve Screen for a Grouped Retrieve Using SQL Join

Figure A-24 shows the Rules Element Retrieve screen:

- The query CARS, DEALERS where CARS.DB_MODEL = DEALERS.DB_MODEL specifies that records from the tables CARS and DEALERS which have common DB_MODEL values are to be combined into one result table.
- As each record in the table CARS is retrieved, the database interface will combine the string my with the value of the field DB_CAR_NAME with the string _A_ and the value of the field DB_MODEL to create an object name. The database interface will then search the knowledge base for an object with this name. Since no object will be found, a dynamic object with this name will be created and linked to the class |car_class|.

- As each object is created values from the database fields DB_MODEL_DATE, DB_PRICE and DB_SPORTIVE in the table CARS will be passed to the property slots Model_date, Price and Sportive. Values from the database field DB_DEALER_NAME in the table DEALERS will be passed to the property slot Dealer_name.

Reference

Field descriptions for this Retrieve operation follow.

Database Type

An Oracle database is being used in this example.

Begin

For most databases this field should be left blank for grouped retrieve operations. Some databases require a statement here. Look up your database type in Chapter Three, "Database Integration Topics" for details.

Query

This field specifies from which table(s) records are to be retrieved. This field can also contain a where clause to limit the records to be retrieved or to specify the criteria used to join two or more tables into one result table.

End

For most relational databases, this field should be left blank. Some databases may require a statement here. Look up your database type in Chapter Three, "Database Integration Topics" for details.

Name

This field specifies that for each record retrieved from the database the string my, the value of the field DB_CAR_NAME, the string _A_ and the value of the field DB_MODEL will be combined to form the object name in which the database values will be stored.

In

This field specifies the objects and/or classes of objects to be searched to determine if an object exists whose name matches the value of the database field(s) specified in the Name field. If this field is left blank, as in this example, then all of the objects in the knowledge base will be searched.

Cursor

This field must be empty for grouped retrieves.

Link To

This field specifies the class to which new objects created by the retrieve are to be linked. In this example, new objects will be linked to the class |cars_class|.

Database Fields / Rules Properties

These columns specify that data from the columns CARS.DB_MODEL_DATE, CARS.DB_PRICE, CARS.DB_SPORTIVE and DEALERS.DB_DEALER_NAME are to be passed to the property slots Model_date, Price, Sportive and Dealer_name. Note that in order to

avoid ambiguity the database field names must be prefixed by the appropriate table name.

Create New Record

This is selected to indicate that if an object with a name specified by the Name field doesn't already exist, it is to be created. If this is not selected, data will only be retrieved into objects which already exist in the knowledge base, and any other records ignored.

Current Forward

This is selected to indicate that the retrieval of any data into property slots will place hypotheses on the agenda according to the RHS forward-chaining strategy currently in effect.

Related Topics

Grouped Retrieve

Database Editor Windows

Access String

Slot Specification for Retrieves

Object Name Specification

Query Retrieve Operations

Also, look up individual arguments and your database type for more detailed information.

Index

Symbols

- @ATOMS 296
- @BEGIN 263
- @CREATE 313
- @CURSOR 268
- @END 282
- @F 288
- @FIELDS 287
- @FILL
 - ADD 265, 266
 - INSERT 310
 - NEW 315
- @FWRD 290
- @NAME 314
- @PROP
 - access string 255
 - Begin field 263
 - End field 282
 - Query field 328
- @PROPS 328
- @QUERY 328
- @SELF
 - access string 255
 - Begin field 263
 - End field 282
 - Query field 281
- @SLOTS 360
- @TYPE 275
- @UNKNOWN 347, 372
- @V
 - access string 255
 - Begin field 263
 - End field 282
 - Link To field 313
 - Query field 281

A

- ABS function 1
- access string
 - environment variables 255
 - interpretations 255
 - pathnames 255
 - specification 254
 - usage 253
- accessing databases 253, 254–256
- ACOS function 2
- actions 2
- AddFile command 211
- agenda 5
- Align Column command 210

- Always Forward field 290
- AND 16
- API 270
- application programming interface vii
- application programming interface *see* API
- arguments
 - keywords 274
 - overview 256–259
- arithmetic operators 331
- ASIN function 6
- AskQuestion Operator 7
- Assign operator 8
- ATAN function 10
- AtomExist Routine 197
- atomic operations
 - atomic retrieves
 - example 396–399
 - specification 259–261
 - atomic writes
 - example 381–383
 - specification 261–263
 - cursor slot 266
 - explicit slots 319
- AtomNameValue Routine 198
- AVERAGE function 11

B

- backward chaining 12
- Backward operator 13
- Begin field 263–265
 - retrieving files 288
- beginning database operations 264
- BOOL2STR function 14
- boolean constants 15
- boolean expressions 16
- boolean formats 17
- Boolean operators 332–333

C

- CEIL function 19
- Center command 210
- CHARFIND function 19
- CharWrap command 210
- classes 20
- column 276
- comment attribute 22
- commit 280, 283
- COMPARE function 23
- comparison operators 24
- ComputeMultiValue Routine 200
- conditions 26
- content filtering 393
- context links 28
- context variable 287

ControlSession Routine 202
 CopyFrame Routine 204
 COS function 29
 COSH function 30
 Create New Record field 265, 338
 Create Object field 266, 281, 286
 CreateObject operator 30
 CreateObjects Routine 205
 CreateReport Routine 207
 Current Forward field 290
 Cursor field 268
 cursor slot

- atomic retrieves 259
- atomic writes 261
- error setting 267
- sequential retrieves 353
- sequential writes 341, 355
- specification 266–268

D

DAL *see* data manipulation language
 data manipulation language 264, 283
 data types 32
 data validation 33
 database editor windows

- arguments overview 256
- description 273

 database interface 268–272
 databases

- access 253, 254
- accessing created files 315
- basics 276, 348, 373
- beginning operations 264
- ending database operations 283
- ending operations 283
- format of data 289
- grouped retrieve 291
- grouped writes 292
- multiple user 295
- multiple-user 295
- range names 263, 282
- return errors 350
- sequential retrieves 352
- sequential writes 354
- supported 275
- unsupported 270
- see also* flat-file databases, relational data-

 bases
 datatype

- conversion 362
- specifying 289

 date 38, 289

- Informix 303
- Ingres 308
- Oracle 302, 325, 326

 Date command 211
 date formats 35
 DATE2FLOAT function 39

DATE2STR function 40
 DAY function 41
 dBase III 277
 DBF3 277
 debugging operations 278–281, 350
 DeleteObject operator 42
 Do Not Forward field 290
 dynamic data exchange (DDE) 43
 dynamic objects 47
 dynamic values *see* @V, @SELF, @PROP

E

End field 280, 282–284
 end of file 267
 environment variables 255
 error messages

- cursor slot setting 267
- general 279
- possible 350
- trapping 279

 error slot 279, 361
 examples

- atomic retrieve 396
- atomic writes 381
- database interface usage 271
- existence filtering 390
- grouped retrieves 383, 387, 393, 407
- grouped writes 375, 378
- sequential retrieves 399, 402

 Excel *see* SYLK
 execute library routines 50
 Execute operator 48
 execute routines 50
 existence filtering

- defined 284–287
- example 390

 EXP function 53
 expressions 54, 287

F

FALSE 15
 field width 318
 fields

- defined 276
- mapping from properties 373
- mapping to properties 348
- selection 314, 319
- specification 287
- width 287

 Fields list 287
 file creation 315
 file retrieves 288
 FileExist Routine 213
 filtering records 296, 338
 filtering retrieves 284, 335
 FindListElem Routine 214

flat-file databases
 access string 254
 atomic retrieves 259
 atomic writes 261
 basics 361
 last record retrieved 267
 opening 276
 query language 329–335
 return errors 350
 Rules Element formats 316
 sequential retrieves 352
 sequential writes 354
 supported 275
 SYLK format 369
 terminology 276
 FLOAT2DATE function 56
 FLOAT2INT function 56
 FLOAT2TIME function 58
 floating point constants 59
 floating point formats 60
 FLOOR function 63
 FOAT2STR function 57
 Footer commands 209
 format attribute 64
 format errors 280
 formats 64, 289–290
 forward chaining 66, 140
 forwarding strategy 290
 functions 334

G

gates 140
 GetListElem Routine 216
 GetMultiValue Routine 218
 GetRelatives Routine 220
 grouped operations
 cursor slot specification 266
 naming objects 314, 320
 naming records 314
 record naming 344
 grouped retrieves
 creating new objects 266
 errors 281
 example 383–386, 387–390, 390–392,
 393–396, 407–410
 existence filtering 284
 linking objects 313
 specification 291–292
 grouped writes
 creating a file 315
 creating new records 265
 example 375–377, 378–381
 inserting a record 310
 object filtering 296
 query operations 338
 record selection 341
 specification 292

H

Header commands 209
 HOUR function 68
 hypotheses 69

I

identifiers 70
 If Change method 71, 294, 295
 In List field 285, 286, 296–297
 Include command 211
 inference 73
 inference priority 74
 inference slot 75
 inference strategy 76
 Informix interface 298–304
 Ingres operations 304–310
 inheritability strategy 79
 inheritance 81
 inheritance priority 82
 inheritance slot 83
 inheritance strategy 84
 InhMethod operator 86
 InhValueDown operator 87
 InhValueUp operator 88
 Init Value attribute 89
 Insert 310
 Insert Only check box 310
 INT2STR function 90
 integer constants 91
 integer conversion 362
 integer formats 91
 interfacing to databases 270
 interpretations 94
 access string 255
 Begin field 263
 End field 282
 In List field 296
 Query field 281
 query language 334
 usage 311
 Interrupt operator 95

J

join example 407
 join operation 336
 Journal Routine 222

K

key see record key
 keywords 273
 knowledge representation features vii
 KNOWN 15

L

LeftAlign command 210
 LENGTH function 96
 LHS conditions 312, 313
 Link To field 313
 LinkMultiValue Routine 223
 LN function 97
 LoadKB operator 98
 LOG function 100
 logical operators 212

M

Margin commands 209
 MAX function 101
 Member operator 102
 Message Routine 225
 meta-slots 103
 Method Editor window

- argument keywords 274
- If Change method 294, 295
- Order of Sources 326, 327

 methods 104
 MIN function 107
 MINUTE function 109
 MOD function 109
 MONTH function 110
 multiple retrieves 327, 352
 multiple user databases 295
 multiple writes 354
 multi-values 111

N

Name field 314

- retrieve operations 280, 320
- write operations 341

 New File field 315
 NewFile command 211
 No operator 112
 NoFormFeed command 211
 NoInherit operator 113
 NOT 16
 NOTKNOWN 15
 NotMember operator 113
 NOW function 114
 null string 254
 NXP file format 316

O

object 115

- as part of slot 319
- creation 266
- filtering 296

object (*continued*)

- linking 313
- naming 314, 319
- updating 284

 operators 331–333
 OR 16
 Oracle operations 255, 321–326
 Order of Sources method 117, 326, 327

P

PageBreak commands 210
 PageLength command 209
 PageWidth command 209
 Parse Routine 227
 password see access string
 pattern matching 296, 313
 pattern matching filtering 286
 PatternMatcher Routine 230
 patterns 120
 portability 269, 275, 339
 POW function 123
 priorities 33, 74, 82
 private slots 150
 PROD function 124
 prompt line attribute 125
 PropagateValue Routine 233
 properties list 328
 property 126
 ProtoDB file format 317
 public slots 150

Q

query

- cannot be processed 267
- errors 279
- filtering example 393, 402
- for flat-file databases 329–335
- for relational databases 335–336
- join operations 336

 Query Arguments field 282
 Query field

- @V 311
- arguments 281
- atomic retrieves 260
- atomic writes 261
- flat-file databases 329
- grouped writes 338
- relational databases 335
- sequential retrieves 353
- statements 328
- where clause 336, 339

 Query Language

- operators 331, 333

 query language

- example 329
- functions 334

- query language (*continued*)
 - interpretations 334
 - operators 330–333
 - values 330
 - wildcards 334
- question window attribute 127
- quotes
 - around interpretations 311
 - in fields 279
- R**
- RAND function 128
- RANDOM function 129
- RANDOMMAX function 130
- RANDOMSEED function 130
- range name 263, 282
- RankList Routine 235
- RDB *see* relational databases
- record keys 341–342
- record naming 314, 344
- records
 - defined 276
 - filtering 284, 345
 - inserting only 310
 - mapping from objects 373
 - mapping to objects 348
 - position 266
 - retrieving multiple 291
 - writing 292, 340
- relational databases
 - access string 255
 - atomic retrieves 259
 - atomic writes 261
 - beginning operations 264
 - context variable 287
 - cursor specification 267
 - datatype specification 362
 - ending operations 283
 - expressions in field names 287
 - field width 287
 - join operations 277
 - query operations 335, 336
 - return errors 350
 - sequential retrieves 352
 - stream number 267
 - supported 275
 - terminology 276
- relational operators 332
- reports
 - logical operators 212
- reserved words 131
- Reset operator 132
- ResetFrame Routine 236
- Retrieve operator 133, 346
- retrieve operator
 - access string 253
 - arguments 256
 - null string 254
- Retrieve window 273
- retrieving
 - dates 289
 - field specification 287
 - files 288
 - forwarding data 290
 - general 348
 - in If Change method 294
 - in LHS conditions 312
 - in Order of Sources method 327
 - in RHS actions 351
 - join operations 336
 - methods 271
 - multiple records 291
 - multiple retrieves 327, 352
 - sequential records 352
 - single record 259
 - slot specification 356–358
 - to constructed slots 319
 - to existing objects 284
 - to explicit slots 319
 - unknown values 347
 - with queries 335, 337
- return errors 350
- RHS actions 351
- RightAlign command 210
- rollback 280, 283
- ROUND function 135
- row 276
- Rule Editor window
 - access string example 254
 - argument keywords 274
 - LHS conditions 312, 313
 - RHS actions 351
- rules 136
- RunTimeValue operator 137
- S**
- SECOND function 138
- SELF 139
- semantic gates 140
- SendMessage operator 141
- sequential operations
 - cursor slot 267, 341
 - explicit slots 319
 - retrieve example 399–402, 402–406
 - retrieves 352
 - writes 340, 354
- Set Column command 210
- SetMultiValue Routine 238
- SetValue Routine 240
- Show operator 145
- SIGN function 148
- sign-on *see* access string
- SIN function 148
- SINH function 149
- slot list 360

slots 150
 constructed names 319, 358, 360
 constructed names example 378
 explicit name 319
 explicit names 356, 359
 for retrieves 356–358
 for writes 358–360
 value changes 295
 spreadsheets *see* flat-file databases
 SQL commit 283
 SQL cursor number 267
 SQL statements 264, 283, 288
 SQL-like queries 329–335
 SQRT function 152
 STDEV function 153
 STR2BOOL function 164
 STR2DATE function 165
 STR2FLOAT function 166
 STR2INT function 167
 STR2TIME function 168
 strategy 154
 Strategy operator 155
 STRCAT function 157
 stream number 267
 STRFIND function 158
 string constants 159
 string formats 160
 string to integer conversions 362
 STRLEN function 162
 STRLOWER function 162
 strong link 140
 STRUPPER function 163
 SUBSTRING function 169
 SUM function 170
 Sybase
 beginning database operations 264
 ending database operations 283
 operations 363–369
 SYLK operations 369
 SYLKDB operations 370
 system attributes 103

T

table 276
 Tabs command 210
 TAN function 171
 TANH function 172
 terminology 276
 TestMultiValue Routine 241
 text file

file commands 211
 AddFile command 211
 Include command 211
 NewFile command 211
 NoFormFeed command 211

text file (*continued*)

screen layout commands 209
 Footer commands 209
 Header commands 209
 LeftMargin command 209
 PageBreak command 210
 PageLength command 209
 PageWidth command 209
 RightMargin command 209
 text commands 210
 Align Column command 210
 Center command 210
 CharWrap command 210
 Date command 211
 LeftAlign command 210
 RightAlign command 210
 Set Column command 210
 Tabs command 210
 WordWrap command 210

text formatting

commands (See also text file)

time 174

time formats 173

TIME2FLOAT function 175

TIME2STRING function 176

Transcript window 278

TRUE 15

U

Unify Routine 248

UNIX 255

Unix

Informix interface 298–304

UNKNOWN 15

unknown values

retrieving 347

writing 372

UnloadKB operator 177

updating records 292

V

value changes 295

Value property 179

VAR function 180

VAX issues 370

VMS 255

W

warning message 289

WEEKDAY function 181

where clause 336, 339

why attribute 182

wildcards 334

WKS operations 370

WordWrap command 210

Write operator 183, 371
write operator
 access string 253
 arguments 258
 null string 254
Write window 273
WriteTo Routine 251
writing
 by key 341
 by position 340
 creating a file 315
 field specification 287
 general 373
 in If Change method 295
 in LHS conditions 313
 in Order of Sources method 327
 in RHS actions 351
 inserting a record 310
 logging slot activity 327
 multiple records 292
 multiple writes 354
 object specification 296
 record specification 340
 sequential records 354
 single record 261
 slot specification 358–360
 unknown values 372
 with queries 337–340

Y

YEAR function 184
YEARDAY function 185
Yes operator 186

*FrameMaker has detected one or more
PostScript errors in this document.
(Jack Godwin)
Please check your output.*

PostScript error (--nostringval--, --nostringval--)